Speed up of the Pydna assembly algorithm

Pedro Queirós[1] and Björn Johansson[1]

[1] Universidade do Minho, Campus de Gualtar, 4710-057, Portugal
pdqueiros@gmail.com

Abstract. Automating and documenting DNA cloning strategies to be used in experiments is a somewhat recent necessity, due to the technologies involved. Still, there are several tools that perform this task, one of them is Pydna. This paper is part of a project for a Master in Bioinformatics and will focus on Pydna and its improvement. Pydna has several modules, this project focused on the module "assembly" which is responsible for the assembly of constructs. Graph-tools, a tool for the manipulation and analysis of graphs in C++, was implemented in Pydna, in place of the currently used NetworkX, which is purely implemented with Python. Another module "simple_paths8", responsible for the creation of both linear and circular construct paths, was also adapted for Graph-tool. Unfortunately, it was concluded that the implementation of Graph-tool wasn't successful in speeding up Pydna, however the profiling of the assembly module contributed to an update of the dseq module, considerably increasing Pydna's efficiency.

Keywords: Pydna, Python, Graph-tools.

1 State of art

1.1 Background

Due to the growing need of creating increasingly complex DNA constructs, various, both in-vivo and in-vitro, DNA assembly protocols have been published. These protocols require the planning of said DNA constructs, which is usually done manually through sequence editors.

While such editors might permit planning a thorough assembly strategy, due to assembly's inherent complexity, they remain prone to human error and thus leading to errors and omissions in the resulting strategies. Despite not always the rule, it's often one finds incompletely documented or ambiguous DNA cloning strategies, which does not allow further consultation for either inspection or to serve as a basis for future projects.

Since the DNA construction process is deterministic there's only one solution for the DNA fragments to combine and one final DNA sequence, as such, it should possible to automate the simulation and proper documentation of these constructs.

1.2 Existing tools

Various tools have been developed to solve this problem, and while providing similar features, their implementation differs in various aspects, and ergo the end result will possibly also be different.

Several tools opt for a graphical user interface, which initially provide an easier experience for new users as it mostly requires the biological know-how of the construction protocol. In general, the user inputs an assembly method and several constructs and is then provided with a protocol which they can use in their experiments. A few of these graphical tools will now be briefly discussed.

Gibthon is a free open source tool that designs primers for use in Gibson Assembly. The last commit was in late 2012 and gibthon.org is no longer available, so it's likely the project has been discontinued. There's also j5 and RavenCAD which offer high level functionality such as optimization of cost and part reuse [1].

Contrasting with previously discussed tools, there's Pydna which is a command line based tool, developed by Björn Johansson, a researcher and assistant professor at University of Minho in Braga, Portugal. This tool will be the main focus of this paper and project.

In comparison, while a graphical user interface might at first be advantageous due to their lower initial learning threshold, they cannot provide the versatility brought forth by a command line based tool. Due to the assembly complex nature, a command line tool might prove more useful in the long term, despite its initial learning "barrier".

2 Project

2.1 What is Pydna? [1]

Pydna is an extensible, free and open source Python library for simulating basic molecular biology DNA unit operations such as restriction digestion, ligation, PCR, automated primer design, Gibson assembly and homologous recombination.

Pydna can aid in the design of DNA constructions and at the same time be a compact, self-contained, unambiguous plan for almost any sub-cloning or DNA assembly experiment.

Pydna allows the mixing of different kinds of assembly protocols with classical restriction endonuclease cut and paste cloning. The execution of the code verifies the accuracy and completeness of the described strategy. All intermediate results are automatically generated and can easily be inspected. Strategies described in Pydna are easy to modify if necessity arises. For instance, a strategy may have to be modified due to for example a particular DNA fragment being refractory to PCR amplification. Pydna would allow simply redesigning primers and reexecute the Pydna code to verify that the strategy and all downstream steps are still correct. A strategy designed by hand would require all steps downstream of the modification to be reassessed.

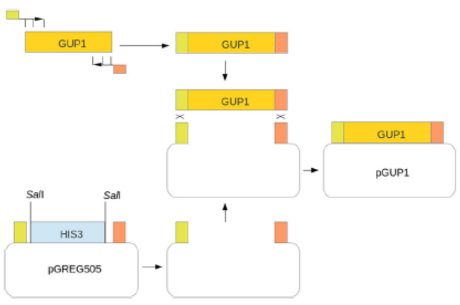
2.2 Main Objective

Pydna was implemented exclusively in Python (the current build uses Python 3.5) and depends mainly on Biopython and NetworkX. It's subdivided in several modules but this project will primarily focus on one particular module, the assembly module. The current implementation of the assembly module algorithm has some issues, most importantly in terms of speed and memory usage. It uses the pure python NetworkX package but there are several faster C implementations. The main objective of this project is to produce an alternative implementation using a graph package (Graph-tools) implemented in C++. A secondary goal is to modify or adapt a new path finding algorithm for returning network paths.

2.3 Pydna in action

To understand how Pydna works and when the assembly module comes into play, here follows an example of the Construction of the pGUP1 vector by homologous recombination (an adaptation from literature by Bosson and coworkers [2]) using Pydna.

We can first see the outline of the cloning strategy below and its implementation with Pydna:



```
import pydna
gb = pydna.Genbank("myemail@mydomain.com")

GUP1rec1sens, GUP1rec2AS = pydna.parse('''
>GUP1rec1sens
gaattcgatatcaagcttatcgataccgatgtcgctgatcagcatcctgtc
>GUP1rec2AS
gacataactaattacatgactcgaggtcgactcagcattttaggtaaattccg
''', ds=False)

pGREG505 = pydna.read("pGREG505.gb")
GUP1_locus = gb.nucleotide("Z72606")
insert = pydna.pcr(GUP1rec1sens, GUP1rec2AS, GUP1_locus)
from Bio.Restriction import SalI
lin_vect, his3_stuffer = pGREG505.cut(SalI)
asm = pydna.Assembly( (lin_vect, insert) )
pGUP1 = asm.circular_products[0]
print pGUP1
```

Fig. 1 Outline and Pydna code of the cloning strategy described for the construction of pGUP1, adapted from [1]

It can be seen how easily Pydna amplifies GUP1 using tailed primers GUP1rec1sens and GUP1rec2AS, proceeds to cut the plasmid pGREG505 with endonuclease SalI and then assembles the plasmid with the GUP1.

2.4 Assembly module

This module provides functions for assembly of sequences by homologous recombination and other related techniques. The assembly algorithm is based on graph theory where each overlapping region forms a node and sequences separating the overlapping regions form edges. It's implemented in three steps:

In the first step, shared homologous subsequences are found by performing a pairwise comparison of all sequences. This is accomplished by using a pure python implementation of the fast suffix array string comparison algorithm by Kärkkäinen and Sanders [1, 3].

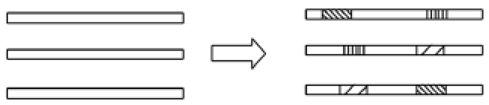


Fig. 2 First step in the assembly, adapted from [1]

In the second step a graph is created where overlapping sequences form nodes and sequences between overlapping sequences nodes form edges. The edges of each linear fragment (5' and 3') are also added as nodes to the graph. The graph capabilities of Pydna are based on the widely used and thoroughly tested NetworkX graph package [1, 4].

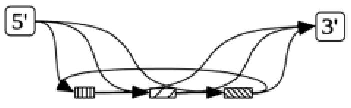


Fig. 3 Second step in the assembly, adapted from [1]

In the third and last step, all possible linear and circular paths are traced through the graph and the sequence of each assembly product is established (this is done by a supporting module-"_simple_paths8.py"). Linear graphs are all graphs between the 5' and 3' edges. In this case, the five sequences share homologous sequences so the resulting graph has two circular sub graphs. All three circular graphs are returned where the largest is the combination of the two smaller sub graphs [1].

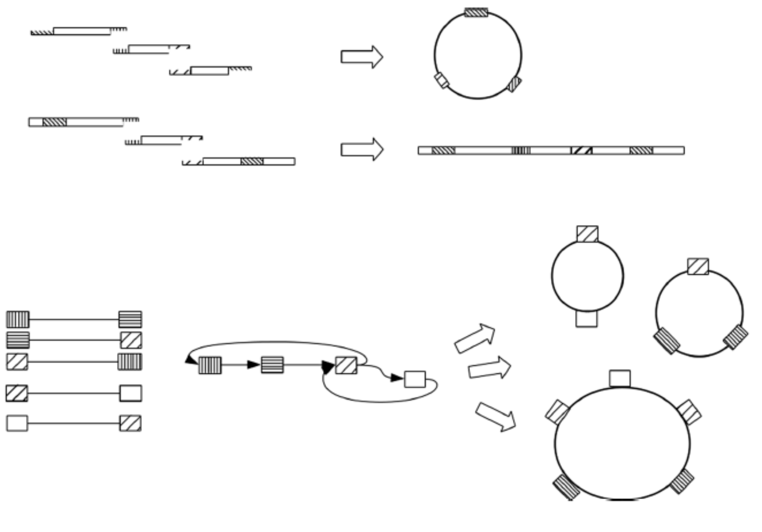


Fig. 4 Third step in the assembly, adapted from [1]

2.5 Graph Algorithm

A graph is a collection of vertices/nodes connected by edges. Graphs can be directed, edges can have values, etc, and their complexity depends entirely on the problem at hand. Due to their versatility, throughout the years, several algorithms that employ graphs have been developed.

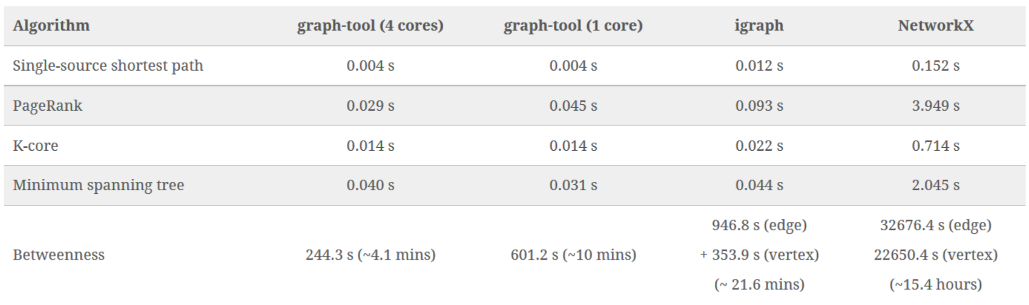
In this project we will mostly deal with networks, which are, in practice, very similar to graphs, a graph has vertices and edges (structure) while a network has nodes and links that hold some kind of information (structure + content). These terms are usually used interchangeably.

Pydna currently uses NetworkX which is package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. More specifically, Pydna uses Networkx’s MultidiGraph class which is a directed graph class that store multiedges (multiple edges between two nodes, where each edge can hold optional data or attributes) and allows self-loops. Any hashable Python object can represent a node, in Pydna’s case these would be the overlapping sequences.

While NetworkX uses a pure python implementation, graph-tools implementation is in C++ which is significantly faster. In Graph-tools, the graph is implemented as an “adjacency list”, where both vertex and edge lists are C++ STL vectors.

As we can see from the table below, NetworkX runs 20 to 170 times slower than graph-tools. Implementing graph-tools in Pydna’s assembly module should increase its efficiency considerably.

Table 1 Graph-tool performance comparison, adapted from [5]



Algorithm	graph-tool (4 cores)	graph-tool (1 core)	igraph	NetworkX
Single-source shortest path	0.004 s	0.004 s	0.012 s	0.152 s
PageRank	0.029 s	0.045 s	0.093 s	3.949 s
K-core	0.014 s	0.014 s	0.022 s	0.714 s
Minimum spanning tree	0.040 s	0.031 s	0.044 s	2.045 s
Betweenness	244.3 s (~4.1 mins)	601.2 s (~10 mins)	946.8 s (edge) + 353.9 s (vertex) (~ 21.6 mins)	32676.4 s (edge) 22650.4 s (vertex) (~15.4 hours)

2.6 Creation of linear and circular network paths

The assembly module is responsible for creating the linear and circular network paths, it does however require another sub-module “_simple_paths8.py”. A secondary goal of this project is to analyze whether the algorithm that finds these paths can be improved, if it can, it’s a secondary goal to do so.

3 Software requirements

3.1 Compatibility of existing packages

Pydna is a cross platform package, meaning that it can be run on Linux, Windows or Mac and its current version requires Python 3.5.

The second tool required for this project is graph-tool, which does not natively support Windows. There are a few packages in Anaconda Cloud which looked promising as they were converted to Windows, however after many attempts, it was concluded that most of these versions were not up to date or were incompatible with either Pydna or the current version of Pydna's background packages. In the end, I decided that the best way to approach this project was to abandon Windows development, install Linux on a separate computer and develop code in this OS.

With compatibility issues out of the way, the installation was then fairly straightforward.

3.2 Installation of main packages

The package manager used in this project was conda, which, allows the installation of packages via console. It's a fairly simple method and it runs smoothly if all packages are compatible.
The installation was achieved with the following lines of code:

```
conda install python=3.5.2
conda config --append channels BjornFJohansson
conda install pydna
conda config --add channels ostrokach-forge
conda config --add channels conda-forge
conda install graph-tool
```

With a few lines of code, all the required packages for both graph-tool and Pydna are installed and both packages should now be usable.

4 Definition of tasks

After meeting with my supervisor Björn Johansson, and exposing the difficulties found previously it was decided that the best approach to this project was as follows:

- Implement Graph-tool exclusively on Linux, whilst forgoing Windows and Mac implementation;
- After implementation, profile the code and test the efficiency of Pydna with Graph-tool.
- Do theoretical research on path-finding algorithms to see if, in the future, it would be possible to improve the current algorithm used in Pydna.

5 Development Stage

5.1 Approach to the problem

As previously discussed, the 2 modules altered for the implementation of graph-tool were the assembly module and the simple_paths module. Changing and adapting these modules to work with graph-tool, while returning the same output was the main objective of my work. It was also essential that Pydna could use NetworkX as a fallback if Graph-tool was not available. This would be the case in most Windows installations of Pydna.

The first step was deciding whether to create a new module that manages the imports dependencies and has the required functions for each case (NetworkX or Graph-tool) or to simply adapt the current modules. Together with Björn, it was decided that the latter approach would be best. I therefore started working and adapting the current code in the assembly and simple_paths modules.

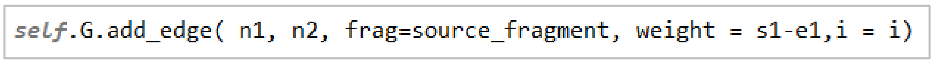
5.2 The particularity of C++

Unlike a dynamic language like Python, C++ is a static programming language, which means that while in Python, types are only known when the program is running, in C++ types are known beforehand and checked for correctness before running the program. So, while a dynamic language might optimize programmer efficiency, by implementing functionality with less code, a static language is designed to optimize hardware efficiency, consequently the code is executed as quickly as possible [6].

This is how graph-tool could contribute to an increase in efficiency. It's also how and why the code between Graph-tool and NetworkX differs.

5.3 Typing of variables

NetworkX works in such a way that you can simply add properties to an edge. After defining the starting node (n1) and target node (n2), this is achieved by adding parameters to the native add_edge function, like so:



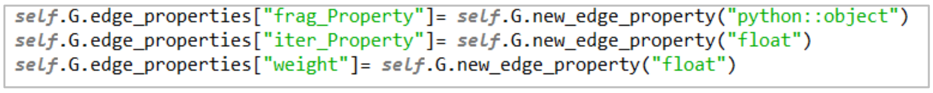
```
self.G.add_edge( n1, n2, frag=source_fragment, weight = s1-e1,i = i)
```

Fig. 5 Adding an edge with NetworkX

These properties can be almost anything, from simple integers or strings to complex python objects. In fact, the "frag parameter" is actually an inherited Dseqrecord object cleaved into a smaller fragment. The "weight" and "i" are simple float objects.

In NetworkX these definitions don't really matter as this package can automatically handle any type of data you send its way. The same can't be said about Graph-tool.

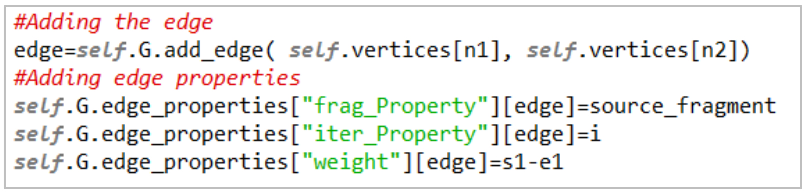
The latter requires the programmer to define what type of data is being sent to the edge, as afterwards graph-tool will send this data to the static language C++. This is accomplished by adding edge properties and their respective data type to a edge properties dictionary.



```
self.G.edge_properties["frag_Property"]= self.G.new_edge_property("python::object")
self.G.edge_properties["iter_Property"]= self.G.new_edge_property("float")
self.G.edge_properties["weight"]= self.G.new_edge_property("float")
```

Fig. 6 Defining edge properties with Graph-tool

Whenever an edge is added we can simply add the properties we wish for:



```
#Adding the edge
edge=self.G.add_edge( self.vertices[n1], self.vertices[n2])
#Adding edge properties
self.G.edge_properties["frag_Property"][edge]=source_fragment
self.G.edge_properties["iter_Property"][edge]=i
self.G.edge_properties["weight"][edge]=s1-e1
```

Fig. 7 Adding edges and their properties with Graph-tool

These changes will allow the creation of graphs with equal information whether it's created by NetworkX or Graph-tool.

5.4 Path-finding algorithms

Pydna's assembly can create linear and circular constructs, hence different algorithms are required.

Both NetworkX and graph-tool offer linear path-finding functions and they actually share the same algorithm, depth-first search (DFS). In NetworkX the function is called "all_simple_paths" and in Graph-tool topology module "all_paths". Since NetworkX apparently uses a modified version of this algorithm, these functions were tested several times to see if their outcome was consistently equal, and, as expected, it was. The time complexity of the DFS algorithm is O(V+E) for a single path but for all the paths in a graph it can possibly scale up to O(V!), where V is the number of vertices/nodes and E the number of edges.

Unlike linear constructs, the algorithm that creates circular constructs is different in the two packages. NetworkX uses a generator version of Johnson's algorithm[7] and Graph-tool uses an algorithm by Hawick & James[8]. The time complexity is identical in both: $O[(V+E)(C+1)]$, V for vertices/nodes, E for edges and C for circuits.

In conclusion, both packages use algorithms with equal time complexity.

5.5 NetworkX as a fallback

Since Graph-tool is not available in Windows (versions untested in Mac) it's necessary to have NetworkX as a fallback in case graph-tool is not available. This is quickly done by checking imports and streamlining the code execution towards a certain path, depending on the graph creation package imported.

6 Profiling and discussion of results

After the implementation of Graph-tool it's necessary, first of all, to check if the output is the same, since Pydna is deterministic. Both packages returned the same output. The second thing to check for is whether there were actual improvements in efficiency. Theoretically there should be, since Graph-tool uses C++, but profiling is obviously still required. All the testing was done on a system with Ubuntu 16.04 LTS, 8 GB RAM and an Intel Core i3-2120.

Initially I did some basic profiling with the package *time*, but alas this can only give us a broad idea and we might need more specific results to evaluate the code. Nonetheless here are the average results, in **seconds,** from several runs (10 to be precise) with 3 and 5 sequences:

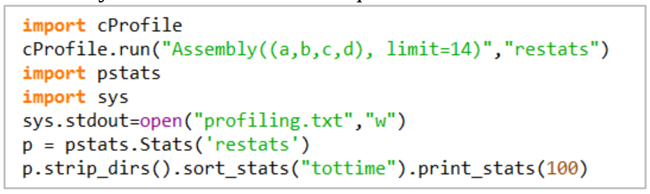
Table 2 Assembly performance

	3 sequences		**5 sequences**	
	NetworkX	*Graph-tool*	*NetworkX*	*Graph-tool*
Graph creation	0,0011	0,0043	0,0024	0,0089
Linear path	0,4231	0,4208	4,1947	4,3563
Circular path	0,0420	0,0425	0,1877	0,2103
Total time	1,0715	1,0623	5,7036	5,9545

Unfortunately, Graph-tool was not really any faster than NetworkX. I expected improvements in the functions that find the linear and circular paths. In a way, this is not a big surprise, since their algorithms have the same time complexity, yet Graph-tool should have come out on top since it uses C++.

To note that the Graph-tool graph creation part of the code was slower. This is probably normal since we are passing complex objects (Dseqrecords) to C++ instead of using python to interpret said objects.

To get a more specific profiling I decided to use cProfile, a commonly used Python code profiler. This analysis was done with 5 sequences:

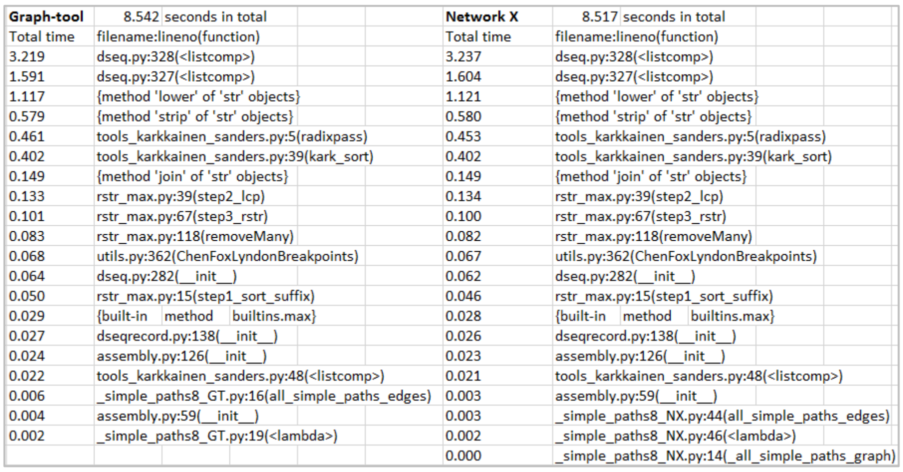


```
import cProfile
cProfile.run("Assembly((a,b,c,d), limit=14)","restats")
import pstats
import sys
sys.stdout=open("profiling.txt","w")
p = pstats.Stats('restats')
p.strip_dirs().sort_stats("tottime").print_stats(100)
```

Fig. 8 Assembly's profilling with cProfile

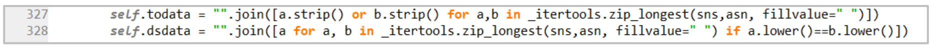
To note that cProfile doesn't actually display real time, but it's still valuable as it allows the measurement of the time spent on each individual function .The results were ordered by total time dedicated to each function and they are as follow:

Table 3 Assembly profiling with cProfile



Graph-tool	8.542 seconds in total	Network X	8.517 seconds in total
Total time	filename:lineno(function)	Total time	filename:lineno(function)
3.219	dseq.py:328(<listcomp>)	3.237	dseq.py:328(<listcomp>)
1.591	dseq.py:327(<listcomp>)	1.604	dseq.py:327(<listcomp>)
1.117	{method 'lower' of 'str' objects}	1.121	{method 'lower' of 'str' objects}
0.579	{method 'strip' of 'str' objects}	0.580	{method 'strip' of 'str' objects}
0.461	tools_karkkainen_sanders.py:5(radixpass)	0.453	tools_karkkainen_sanders.py:5(radixpass)
0.402	tools_karkkainen_sanders.py:39(kark_sort)	0.402	tools_karkkainen_sanders.py:39(kark_sort)
0.149	{method 'join' of 'str' objects}	0.149	{method 'join' of 'str' objects}
0.133	rstr_max.py:39(step2_lcp)	0.134	rstr_max.py:39(step2_lcp)
0.101	rstr_max.py:67(step3_rstr)	0.100	rstr_max.py:67(step3_rstr)
0.083	rstr_max.py:118(removeMany)	0.082	rstr_max.py:118(removeMany)
0.068	utils.py:362(ChenFoxLyndonBreakpoints)	0.067	utils.py:362(ChenFoxLyndonBreakpoints)
0.064	dseq.py:282(__init__)	0.062	dseq.py:282(__init__)
0.050	rstr_max.py:15(step1_sort_suffix)	0.046	rstr_max.py:15(step1_sort_suffix)
0.029	{built-in method builtins.max}	0.028	{built-in method builtins.max}
0.027	dseqrecord.py:138(__init__)	0.026	dseqrecord.py:138(__init__)
0.024	assembly.py:126(__init__)	0.023	assembly.py:126(__init__)
0.022	tools_karkkainen_sanders.py:48(<listcomp>)	0.021	tools_karkkainen_sanders.py:48(<listcomp>)
0.006	_simple_paths8_GT.py:16(all_simple_paths_edges)	0.003	assembly.py:59(__init__)
0.004	assembly.py:59(__init__)	0.003	_simple_paths8_NX.py:44(all_simple_paths_edges)
0.002	_simple_paths8_GT.py:19(<lambda>)	0.002	_simple_paths8_NX.py:46(<lambda>)
		0.000	_simple_paths8_NX.py:14(_all_simple_paths_graph)

We can conclude that both Graph-tool and NetworkX are almost identical, taking very little time and that the inefficiency comes from dseq.py, another module within Pydna. Below dseq.py we can also find that several methods for string objects are taking a large proportion of total execution time. In truth, if we analyze the code in dseq.py we can see that these methods are actually being called within this very same module.



```
327         self.todata = "".join([a.strip() or b.strip() for a,b in _itertools.zip_longest(sns,asn, fillvalue=" ")])
328         self.dsdata = "".join([a for a, b in _itertools.zip_longest(sns,asn, fillvalue=" ") if a.lower()==b.lower()])
```

Fig. 9 dseq.py lines 327 and 328

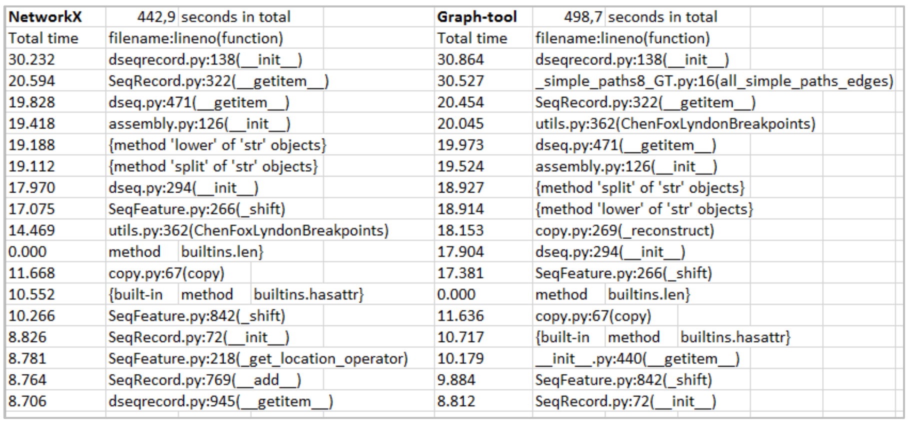
After talking with Björn, dseq.py was improved and efficiency was re-tested.

Table 4 Assembly performance after changing dseq.py

	3 sequences		**5 sequences**		**10 sequences**	
	NetworkX	*Graph-tool*	*NetworkX*	*Graph-tool*	*NetworkX*	*Graph-tool*
Graph creation	0,0011	0,0109	0,0027	0,0092	0,0070	0,0256
Linear path	0,0227	0,0312	0,2308	0,2521	292,3644	336,8903
Circular path	0,0165	0,0194	0,0707	0,0700	15,4016	23,4506
Total time	0,6659	0,6745	1,6360	1,6244	310,8606	363,4742

As we can see, there were significant improvements in efficiency. This time around, with an optimized dseq module, I also ran an average on the assembly time of 10 sequences, to test whether there was any exponential improvement with Graph-tool. There wasn't. We can see that with a higher number of sequences, NetworkX was actually a lot faster than Graph-tool. To understand the difference with 10 sequences I ran a cProfile, with 5 and 10 sequences. Since the performance was almost identical with 5 sequences, I'm only analyzing the cProfile with 10 sequences. This, in an attempt to understand why Graph-tool is slower despite using C++.

Table 5 Assembly profiling with cProfile after changing dseq



NetworkX	442,9 seconds in total	**Graph-tool**	498,7 seconds in total
Total time	filename:lineno(function)	Total time	filename:lineno(function)
30.232	dseqrecord.py:138(__init__)	30.864	dseqrecord.py:138(__init__)
20.594	SeqRecord.py:322(__getitem__)	30.527	_simple_paths8_GT.py:16(all_simple_paths_edges)
19.828	dseq.py:471(__getitem__)	20.454	SeqRecord.py:322(__getitem__)
19.418	assembly.py:126(__init__)	20.045	utils.py:362(ChenFoxLyndonBreakpoints)
19.188	{method 'lower' of 'str' objects}	19.973	dseq.py:471(__getitem__)
19.112	{method 'split' of 'str' objects}	19.524	assembly.py:126(__init__)
17.970	dseq.py:294(__init__)	18.927	{method 'split' of 'str' objects}
17.075	SeqFeature.py:266(_shift)	18.914	{method 'lower' of 'str' objects}
14.469	utils.py:362(ChenFoxLyndonBreakpoints)	18.153	copy.py:269(_reconstruct)
0.000	method builtins.len}	17.904	dseq.py:294(__init__)
11.668	copy.py:67(copy)	17.381	SeqFeature.py:266(_shift)
10.552	{built-in method builtins.hasattr}	0.000	method builtins.len}
10.266	SeqFeature.py:842(_shift)	11.636	copy.py:67(copy)
8.826	SeqRecord.py:72(__init__)	10.717	{built-in method builtins.hasattr}
8.781	SeqFeature.py:218(_get_location_operator)	10.179	__init__.py:440(__getitem__)
8.764	SeqRecord.py:769(__add__)	9.884	SeqFeature.py:842(_shift)
8.706	dseqrecord.py:945(__getitem__)	8.812	SeqRecord.py:72(__init__)

What I found was consistent with the results and data I'd gathered and researched previously. dseq.py remains a time-consuming module, despite improvements, and Pydna doesn't really benefit from Graph-tool capabilities. The moments where graph algorithms are used are very "niche" and in fact, by "translating" complex objects between two languages, we are only encumbering (as seen from the 30 seconds in _simple_paths8_GT) an already optimized process. Which is why NetworkX is more efficient, seeing that it can access the fragment objects directly.

7 Conclusion

Unfortunately Graph-tool implementation hasn't proven to be beneficial to Pydna, because, ultimately, Pydna's execution time doesn't really suffer in the assembly module. In fact, NetworkX is actually faster than Graph-tool when used for Pydna's necessities.

In my opinion, retaining NetworkX as the sole graph analyzing tool would be best, as it has proven to be more efficient than Graph-tool, whilst having the benefit of being a cross-platform package, a trait shared with Pydna. NetworkX is also more intuitive and produces a more readable code.

In retrospective, while I was not able to improve Pydna by implementing Graph-tool, by profiling and analyzing Pydna's performance I've managed to gather some info that could prove useful in the future. Pydna's efficiency might still have some room for improvement, just not with the implementation of Graph-tool.

8 Supplemental material

I have included several files alongside this report:

- The code developed;
- The time evaluation for 3, 5 sequences before changing dseq.py;
- The time evaluation for 3, 5 and 10 sequences after changing dseq.py;
- The profiling before changing dseq.py for 5 sequences with cProfile;
- The profiling after changing dseq.py for 5 and 10 sequences with cProfile;
- Profiling and timing (average of 10) in a more readable format (.xlsx).

All this material is currently available on Github: https://github.com/PedroMTQ

9 Acknowledgements

I'd like to acknowledge the authors of the paper "Pydna: a simulation and documentation tool for DNA assembly strategies using python." [1]. Their paper was crucial, both for the comprehension of Pydna as well as the development of my work and creation of this report. I'd also like to thank Björn Johansson for supervising my work.

References

1. Pereira F, Azevedo F, Carvalho A, Ribeiro GF, Budde MW, Johansson B. Pydna: a simulation and documentation tool for DNA assembly strategies using python. BMC bioinformatics. 2015; 16:142.
2. Bosson R, Jaquenoud M, Conzelmann A. GUP1 of Saccharomyces cerevisiae encodes an O-acyltransferase involved in remodeling of the GPI anchor. Molecular biology of the cell. 2006; 17(6):2636-45.
3. Kärkkäinen J, Sanders P. Simple Linear Work Suffix Array Construction. In: Baeten JCM, Lenstra JK, Parrow J, Woeginger GJ, editores. Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30 – July 4, 2003 Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg; 2003. p. 943-55.
4. Hagberg AA SD, Swart PJ. Exploring network structure, dynamics, and function using NetworkX. In: Proceedings of the 7th python in science conference. SciPy2008. 2008(Pasadena, CA USA: 2008):11-5.
5. Graph-tool performance comparison 2015. Available in: https://graph-tool.skewed.de/performance
6. Pepersack J. Differences between a dynamic programming language and a static programming language . 2015. Available in: https://www.quora.com/What-are-the-differences-between-a-dynamic-programming-language-and-a-static-programming-language-Which-one-is-better-or-in-other-words-has-a-better-prospect.
7. Johnson DB. Finding All the Elementary Circuits of a Directed Graph. SIAM Journal on Computing. 1975; 4(1):77-84.
8. James KAHaHA. Enumerating Circuits and Loops in Graphs with Self-Arcs and Multiple-Arcs. Proceedings of FCS. 2008:14-28.