Evolutionary Algorithms for Maximum Matchings

Alexander van Renen

03.09.2011

Contents

1 Introduction

Charles Darwin described in his book `On the Origin of Species` in 1859 the principle of evolution. Evolution is the process of improving the adaptedness of an animal to its environment by a combination of mutation, recombination and selection. In the 1960s computer scientists started using this great technique for optimization problems. What it basically does is the following. It starts with a set of randomly chosen solutions for a problem, then it picks the best solutions using a so called fitness function and combines them to form the next set of solutions (or next generation). Just like in nature, this is iterated until the solution is good enough.

Besides the evolutionary algorithms which use the principle of evolution, other techniques have been adapted from nature. The so called Ant Colony Optimization approach is inspired by the way ants uses pheromones to find shortest paths between the colony and food places. Other examples would be particle swarm optimization or artificial immune systems. All of these techniques belong to the field of bio-inspired computation.

The great thing about bio-inspired computation is that nearly no problem specific knowledge is needed to solve the problem (except the fitness function). Which makes these algorithms very adaptive and consequently a good approach to solve new problems. In case of well studied problems the bio-inspired algorithms are usually slower than the classic ones. But for very complex problems like in computational biology or engineering these algorithms are very attractive. For example in complex engineering bio-inspired algorithms are used for problems, whose structure is not known. In this cases a specific input can only be evaluated or tested by performing a real life experiment or by running a complex simulation. These problems are called black box optimization problems and bio-inspired algorithms have been shown to be very efficient on these problems.

The biggest advantage of bio-inspired algorithms is their adaptability to every problem. Unfortunately this results in a problem: By using nearly no problem specific knowledge most of the algorithms choices are done using random decisions, which makes the analysis very hard. The first papers on upper runtime bounds for simple evolutionary algorithms were published in 1992 by Muehlenbein.

On the following pages we will focus on evolutionary algorithms and show how they can be applied to the maximum matching problem in graph theory.

2 Maximum matchings

In this section we will shortly recap some general knowledge about graphs, focusing on maximum matchings. This section will also introduce the notations, we am going to use.

2.1 About graphs

There are two kinds of graphs: directed and undirected graphs. With this paper being about maximum matchings we will just need undirected graphs. An undirected graph G is formally defined as a tuple:

$$G = (V, E)$$

The first component V is a finite set representing all vertices in the graph. The second component E represents the edges of the graph, this is done by a set of unordered pairs. So if the vertex u and v should be connected in the graph, we just add the pair $\{u, v\}$ to the edge set E. Here is a simple example (see figure 1 for a visualisation):

$$G = (V, E)$$
$$V = \{1, 2, 3, 4, 5\}$$
$$E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{4, 5\}, \{3, 5\}\}$$

This way of representing a graph is call adjacency list, its a very basic technique. But during the construction of the algorithm we will find it to be very helpful. Please note that there are other more advanced techniques to represent a graph, like storing it in an adjacency matrix (which make it easy to find all neighbors of a vertex).

2.2 Matchings

As we want to create an evolutionary algorithm for maximum matchings, we will have a look at what a maximum matching is. Therefore we will first define a matching.

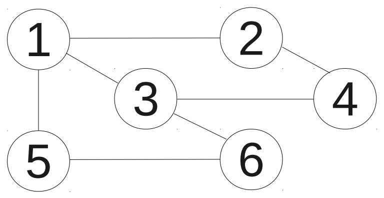

Figure 1: Simple graph example.

2.2.1 Matching in graphs

A matching of a graph is a subset of its edge set. Each edge in the matching has to have its own start and end vertex. Or in other words: No edge of the matchings shares a common node. Formally a matching M of a graph $G = (V, E)$ is defined like this:

$$M \subseteq E$$

$$\forall\ a = \{u, v\}\ \in M.\ \neg\exists\ b = \{r, s\} \in M \backslash \{a\}.\ r = u \lor r = v \lor s = u \lor s = v$$

See Figure 2 for some examples, the edges of the matching are in green.

2.2.2 Maximum matchings

Knowing what a matching is, we can now define a maximum matching. A matching is a maximum matching if there is no other matching with a higher number of edges. To avoid confusions we would like to clarify the term maximal matching. A matching is a maximal matching if it is not possible to add another edge to the matching. A maximum matching is a global maximum considering the number of edges, where a maximal matching is just a local maximum. Each maximum matching is a maximal matching. See Figure 3 for maximal matching examples and Figure 4 for maximum matchings.

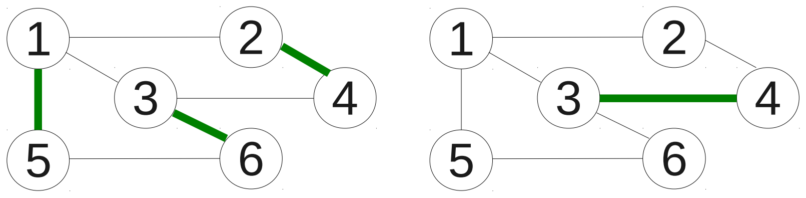

Figure 2: example for a matching.

2.3 Runtime

This problem has been studied for quite a long time, as it is very important in many applications, as we will see in the next part. In 1980 Micali and Vazirani came up with an algorithm that solves this problem in time $O(\sqrt{|V|} \cdot |E|)$.

2.4 Area of application

Maximum matchings are used as well for finding matchings between two sets. This problem arises for example when planing the schedule for the lecture rooms at a university. Or one could use it to assign the topics for presentations to students when planing a trip to Sarntal.

3 Evolutionary algorithms

In this composition we would like to introduce the (1+1) EA algorithm. At first we will have a look at the general concept of evolutionary algorithms and learn how they work. Then we will introduce the (1+1) EA and explain how it can be applied to the maximum matching problem.

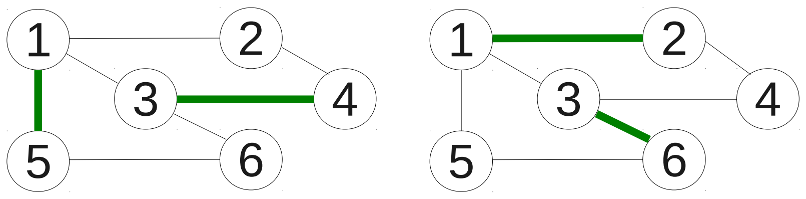

Figure 3: maximal matching examples.

3.1 The general concept

An evolutionary algorithm solves a problem. The problem can be formalized by a triple (S, f, Ω). Where S is the search space, f is the fitness function and Ω is a set of constraints.

The search space contains all possible solutions. Considering the optimization of a simple function $g : \mathbb{R} \rightarrow \mathbb{R},\ g(x) = -x^2$ the search space S would be the set of all real numbers. The fitness function evaluates an element of the search space, in terms of how good the solution is. Ω contains a set of constrains, like x is not allowed to be zero.

In each step an evolutionary algorithm has a so called population (which is a set of search points). This population produces an offspring population, which becomes the new population in the next step. To produce the offspring population a so called variation operators are used.

The most important variation operators are the mutation and the crossover operator. Usually the crossover operator is used first to produce the offspring population, then the mutation operator is used on some of the new elements from the offspring. To be able to design good variation operators it is important to chose a good representation for the elements of the search space (we will see what good means when we design our algorithm)

As you might have noticed there are three things to do if you apply an evolutionary algorithms to a specific problem.

- Chose a representation for the elements of the search space.

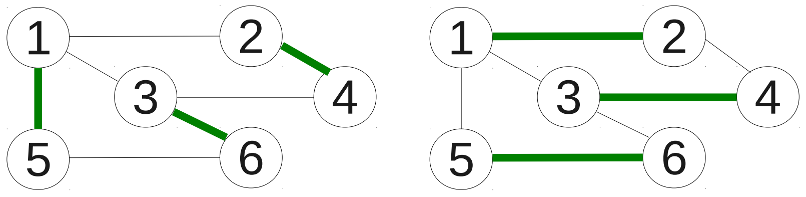

Figure 4: maximum matching examples.

- Create a fitness function.
- Design the variation operators.

3.2 The (1+1)EA

Lets just have a look at the algorithm itself:

1. Choose $s \in \{0,1\}^n$ uniformly at random.
2. Produce s' by flipping each bit of s independently of the other bits with probability $1/n$.
3. Replace s with s' if $f(s') \geq f(s)$.
4. Repeat Steps 2 and 3 forever.

You might have noticed that the first population is created in the first step by random, but this is good enough. In the case of the (1+1) EA all generations have just one member. In the second step we apply a mutation operator. Then we evaluate the results using the fitness function in step three, if the new population is fitter than the old one we apply this change. This concept (mutation and applying changes if they are good) is usually not repeated forever, but until the result is good enough.

In the following section we will apply this algorithm to the maximum matching problem.

3.2.1 Representation

At first we have to chose a good representation of the elements of the search space. In the case of maximum matchings the search space consists of all subsets of the edge set of the graph. A straight forward representation would be to use a bit string $b = (b_1, b_2, .., b_m) \in \{0,1\}^m$ where m is the number of edges. If the i-th bit of the string is set it means that the i-th edge of the graph is a member of the matching. Otherwise the i-th edge of the graph is no member of the matching. (Note: the representation allows sets of edges which are no real matchings, this issue will be solved by the fitness function).

This representation is actually a pretty good one. This is because in the (1+1) EA the mutation operator (step 2 of the algorithm) flips random bits. Such a bit flip should only result in local small changes of the result, otherwise it is likely to get a complete different graph in one generation.

3.2.2 Fitness function

Now lets chose a fitness function. This function should map a point of the search space to a real number, the higher the number the better the point of the search space. In the case of maximum matchings a point is good if the matching contains many edges. In our representation this is reflected by many 1's in the bit string (remember a 1 in the bit string means that this edge is contained in the matching). Consequently a good idea for a fitness function would be to sum up all the ones of the bit string.

$$f(s) = \sum_{i=1}^{m} s_i$$

But this leaves the problem that results that are no matchings are still accepted, so we need a little trick. We do this by introducing a second function the so called penalty function p. The function p for a vertex takes a vertex of the graph as an argument and returns the number of adjacency edges in the matching (this is denoted with $d(v)$) minus one.

$$p(v) = d(v) - 1$$

For a search point s we define the penalty function as the sum of the results of the penalty functions for all vertices of the graph.

$$p(s) = \sum_{v \in V} p(v)$$

Now we add this penalty function to the fitness function.

$$f'(s) = (-p(s), f(s))$$

This function has to be maximized in lexicographical order.

3.2.3 Variation operator

The variation operator is specified by the (1+1) EA, so there is not much work for us to do here. For other EAs a countless number of variation operators exists. Now we have got every part we need to implement the algorithm.

3.3 Approximation quality

Classical algorithms are designed to be efficient and to provide provable runtime boundaries. In Contrast bio-inspired algorithms are designed to be wildly applyable, which leads to algorithms that use a lot of random choices, which makes them very hard to be analyzed. The following section will present some results on approximation quality and runtime bounds.
We will prove that the earlier presented (1+1) EA is able to find a good approximation of a perfect solution, in polynomial time. This is done in two phases, at first we will show that a valid search point is found fast. Then we will show that starting from a valid search point the algorithm finds a good approximation in polynomial time.

Lemma 1 (Valid matching)**.** *A (1+1) EA is able to find a valid matching in expected time $\Omega(m \cdot log\ m)$.*

Proof. A search point is a valid matching as soon as the first parameter of the result of the fitness function is positive. Consequently the second argument is not important, so let $k := -p(s)$ (the first parameter) and let $m := |E|$. Obviously $k \leq 2m$ holds, because if we add all edges to the matching each edge is at most counted twice. From this we can follow that there are at least $\lceil k/2 \rceil$ edges chosen by the search point s. If we remove one of this edges k will be decreased. The probability to remove a specific edge is exactly $1/m$. Hence the expected waiting time is bound by $\Omega(m/k)$ (there are k different edges we can remove). Now we will sum up over all search points $1 \leq k < 2m$:

$$\sum_{k=1}^{2m} 1/k = \Omega(ln\ m) \quad \textit{using the harmonic series}$$

yields to the claim: $\Omega(m\ log\ m)$ □

The first step is done, now we have to show that one can find a good approximation starting from the valid search point in expected polynomial time. For this we need the notion of an augmenting path, which is defined as follows:

Definition 1 (Augmenting path)**.** *A path through the vertices $v_1, .., v_{k+1}$ is called an augmenting path with respect to a matching M if:*

- *The edges $\{v_{2i}, v_{2i+1}\}, 1 \leq i \leq k/2$ belong to the matching.*
- *The other edges do not belong to the matching.*
- *The vertex v_1 and v_{k+1} are free vertices. A free vertex is not touched by any edge from the matching.*

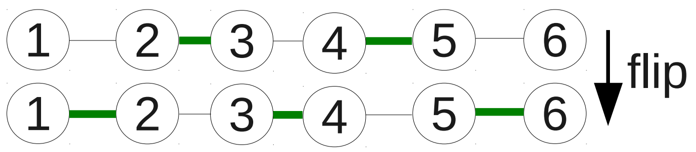

Figure 5: Example for an augmenting path.

Hopcraft and Karp have shown in 1973 the following theorem [NW10]:

Theorem 1 (Hopcraft and Karp)**.** *A matching is of maximum cardinality if and only if there exists no augmenting path with respect to the matching.*

Consequently we will find a perfect solution if we eliminate all augmenting paths. It should be clear that the probability to flip a complete augmenting gets lower, as its length increases. We will shows that one can obtain an upper bound for the length of the shortest augmenting path in a certain matching in the next lemma. Then we will use this bound to prove an upper bound for the amount of time needed to improve the matching by one edge.

Lemma 2 (Augmenting path length)**.** *Let $G = (V, E)$ be a graph, M a non-maximum matching, and M^* a maximum matching. Then there exists an augmenting path with respect to M whose length is bounded from above by $L := 2\lfloor |M|/(|M^*| - |M|)\rfloor + 1$.*

Proof. Let M' be any maximum matching and $G' = (V, E')$ with $E' := M \otimes M'$, where $\otimes$ denotes the symmetric difference. Consequently G' consists of these edges which are just included in one of the two matchings, it represent the difference between the matchings. If it would be empty both would be maximum matchings. There are just vertex-disjoint circle and paths in G', because it is not possible that one vertex has 3 or more adjacency edges (considering the properties of a matching). Now we will have a individual look at each of these circles and paths, with a focus on what it represents for the matchings.

- Even Circles: In a circle the number of edges from G is equal to the number of edges from G', hence this encodes no possibility for an improvement of the matching M, because flipping each edge would just change the matching not improve it.
- Odd Circles: There are no odd circles, otherwise two edges from M or M^* would be adjacency.
- Even paths: Same situation as in even circles, flipping each edge just changes the matching.
- Odd paths: In an odd path the number of edges differs by one. These paths represent an augmenting path for the matching with fewer edges, consequently the number of edges from M^* is one higher then the number of edges from M (because M^* is already maximum).

With each odd path representing a difference of 1 between the two matchings, there have to exist $|M^*| - |M|$ odd paths. (Note that an odd path is allowed to have the length one, representing a augmenting path of length zero, representing a free edge). The pigeonhole principle yields that there exists at least one odd path with at most $\lceil |M|/(|M'| - |M|) \rceil$ from M. This path has $L := 2\lfloor |M|/(|M^*| - |M|) \rfloor + 1$ from M and M^* together, yielding the claim. □

Now we have gathered every tool we needed to prove, what we were going for.

Theorem 2 (Approximation Quality). *The (1+1) EA finds a $(1+\epsilon)$-optimal solution in expected time $O(m^{2\lceil 1/\varepsilon \rceil})$, with $\epsilon > 0$.*

Proof. We know that a valid matching is found in $\Omega(m \cdot log\ m)$, which is also in $O(m^{2\lceil 1/\varepsilon \rceil})$. Consequently we can assume that we start from a search point, which represents already a valid matching. Let M^* be a maximum matching then M is a $(1+\epsilon)$-optimal solution if $(1+\epsilon)\cdot|M| \geq |M^*|$. If we have found such a solution M we are ready, otherwise we know that,

$$(1+\epsilon)\cdot|M| < |M^*|$$

, and

$$\frac{|M|}{|M^*|-|M|} < e^{-1}$$

, and consequently:

$$\left\lfloor \frac{|M|}{|M^*|-|M|} \right\rfloor \leq \begin{cases} \lfloor e^{-1} \rfloor = \lceil e^{-1} \rceil - 1 & \text{if } e^{-1} \text{ is not an integer} \\ \lfloor e^{-1} \rfloor - 1 = \lceil e^{-1} \rceil - 1 & \text{if } e^{-1} \text{ is an integer} \end{cases}$$

holds. If we insert this into $L = 2\lfloor |M|/(|M^*|-|M|) \rfloor + 1$, which we have already proven, we get $L \leq 2\lceil 1/\epsilon \rceil - 1$ as an upper bound for the augmenting path length in a specific step. Let $l \leq L$ then the probability that we flip the edges of this augmenting path (with length l) is $(1/m)^l(1-1/m)^{m-l} = \theta(m^{-l})$. Hence the expected waiting time for the flip of one augmenting path and thus the improvement of the matching by one edge is $\theta(m^l)$. It is sufficient to perform such a step $m \leq |M^*|$ times. Using this and replacing l we get $O(m^{2\lceil 1/\epsilon \rceil})$. □

Now we are nearly ready, the only problem left is that we just showed expected waiting times. Using the Markov's inequality one can solve this issue by introducing a constant c such that the previous theorem holds for the bound $c \cdot m^{2\lceil 1/\epsilon \rceil}$. One obtains the final corollary for this section.

Corollary 1 (Approximation quality). *If we run (1+1) EA for $4cm^{2\lceil 1/\varepsilon \rceil}$ iterations, we obtain a polynomial-time randomized approximation scheme for the maximum matching problem, i.e., independently of the choice of the first search point, the probability of producing a $(1+\epsilon)$ optimal solution is at least $3/4$.*

This is a great result, because it shows that the (1+1) EA for maximum matchings find a $(1+\epsilon)$ optimal solution in a fixed number of steps, with a certain probability.

3.4 Upper bounds for optimal solutions on paths

In this section it will be shown how to prove an upper runtime bound for an optimal solution for a path. Its easy to see that a maximum matching for such a graph consists of $\lfloor m/2 \rfloor$ edges.
Before we start with the proof we have to define what a Prelevant step. A Prelevant step is a step, which alters the shortest augmenting path P of the current matching. Let $E(R)$ denote the number of expected Prelevant steps and $E(T)$ the expected number of total steps. Then $E(T) \leq E(R) \cdot p^{\lceil} - 1$, where p is the probability for a Prelevant step.

Theorem 3 (Optimal solution for a path)**.** *For a path of m edges, the expected optimization time of the (1+1) EA is $O(m^4)$.*

Proof. In the previous section we have already shown that the expected time to find a valid matching is in $\Omega(m \cdot logm)$, which is captured by $O(m^4)$.

Let the size of the current matching be $\lceil m/2 \rceil - i$ then we know that there exists at least i augmenting paths and one of length at most $l := m/i$, we call that one P. The probability to alter P and thus the probability for a Prelevant step is $\Omega(1/m^2)$. (this is because we have to flip at most two edges to alter P). Assuming a number of $O(l^2)$ Prelevant steps is sufficient to improve the matching by one edge, then $\sum_{i=1}^{\lceil m/2 \rceil} O((m/i)^2) = O(m^2)$ Prelevant steps are enough to find a optimal solution. Consequently the expected optimization time is in $O(m^4)$.
So we just need to show that $O(l^2)$ Prelevant steps are sufficient to improve the matching by one edge. This is basically done by introducing so-called Pclean steps, which are in principe Prelevant steps which have only local changes. Then one shows that a phase of $O(m^2)$ Prelevant steps contains only Pclean steps with a probability of $\Omega(1)$, which means that the algorithm does only small changes most of the time. This is done by looking at each group of possibilities. Then we assume (pessimistically) that a shortening shortens the pace always by two edges and furthermore that the probability for that is exactly $1/2$. This reduces the problem to a fair random walk, hence an expected number of $O(l^2)$ Prelevant steps reduces the length of P to at most one. (shown by using Markov's inequality). If the length is zero the matching is improved by one edge and we are ready. Otherwise a free edge exists and consequently a step is Prelevant with probability $O(1/m)$, hence, the next Prelevant step improves the matching with probability $O(1)$. Now we have shown that a number of $O(l^2)$ Prelevant steps is sufficient

to improve the matching by one edge. This was what we assumed at the beginning and now having that shown we are done. □

The reason why this bound is so high is that we are dealing with a blind local search, the algorithm does not know where to go and will produce a lot of invalid search points. Nevertheless the paths is still an easy example because the augmenting path is easy and there is only one way to lengthen it, which is on general graphs not the case.
Polynomial time boundaries like this one can be proven on some other simple graph classes.

3.5 Upper bounds for optimal solutions on general graphs

In this last section on runtime boundaries we will prove a lower bound for the (1+1)EA on general graph types. To prove a lower bound we have to consider the worst possible graph. Such a graph was found by Sasakik and Hajek in 1988 and used by Giel and Wegener (2003, 2006) for this poof. This graph is best explained by a simple picture (see 6).

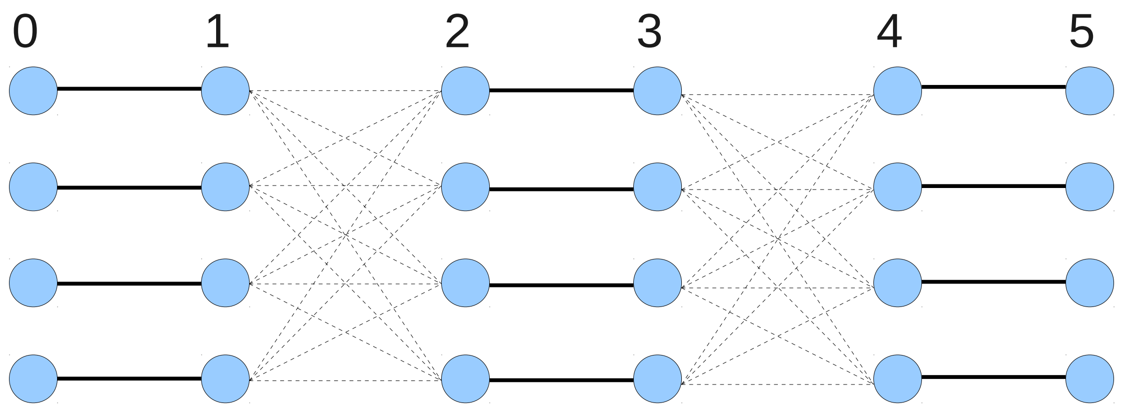

Figure 6: The $G_{4,4}$ is an worst case graph for the maximum matching problem.

The indexes h and l of the Graph $G_{h,l}$ denote the number of rows h and the number of columns $l+1$. One can show that the (1+1)EA has an exponential expected waiting time to find a optimal solution for this graph. We will just outline the idea of the prove here. // To prove this one assumes that the matching already is an almost perfect matching. An almost perfect matching is a matching, with only one edge less than a maximum matching. Consequently there exists only one augmenting path. This augmenting path has two properties:

- The augmenting path goes from left to right, this means that it contains at most one edge from each column.
- There are two shortenings and $2h$ lengthenings for the path. Except the path is at the left or right end of the graph, then there are h lengthenings and two shortenings.

In the proof one shows at first that such a situation (with one augmenting path) will occur with a probability of $\Omega(1)$. Then one reduces this problem to a (really) unfair random walk (with probability $2/(2+2h)$ to shorten the path and $2h/(2+2h)$ to lengthen it). This leads to an expected exponential waiting for adding the last edge to the matching.

4 Implementation

Now having talked for a while about theoretical issues, lets praxis and implement this algorithm and get some test data. I was able to write the algorithm in c++ in about an hour. I create a structure to store the graph, which offers a fitness function, which evaluates the fitness of a given string $s \in 0,1^m$. Having done this I was nearly ready, the only thing left to do was inserting the graph data and changing the pseudo code from above to real code. This looks like this:

```
   // step one - choose random string in {0,1}^m
   Graph g("ulysses16.xml");

%    vector<uint32_t> s(m);
   for(uint32_t i=0; i<m; i++)
      s[i] = random()%2;
```

```
for(uint32_t count=0; count<iterations; count++) {

   // step two - produce s2 by flipping each bit of s with proability 1/m
   vector<uint32_t> s2 = s;
   for(uint32_t i=0; i<m; i++)
      if(random()%m == 0)
         s2[i] = (s[i]==0)?1:0;

   // step three - replace s by s2 if f(s2) <= f(s)
   if(g.f(s2) >= g.f(s))
      s = s2;
}
```

Which is really easy and short as you can see. Nevertheless it archives an optimal solution for the graph from the first part of this paper in only 29 iterations. Moreover the complete code, including the part for the graph is just about 100 lines. A classic algorithm for this problem wont be that easy to implement. Consider that the (1+1) EA is the most basic algorithm in the field of evolutionary algorithms and one can archive better results with more complex EAs. //

5 Conclusion

Consequently one can say that bio-inspired algorithms are in many cases worth considering, especially in new areas and complex areas. We have shown that one can prove that an $(1+\epsilon)$ optimal solution is found in polynomial time. Moreover we have shown that it is also possible to prove that the (1+1)EA is able to find optimal solutions on simple graph classes like a path. Nevertheless

References

[NW10] Frank Neumann and Carsten Witt. *Bioinspired Computation in Combinatorial Optimization - Algorithms and Their Computational Complexity.* Springer, 2010.