A Measure & Conquer Approach for the Analysis of Exact Algorithms

Fabrizio Grandoni*

May 19, 2009

1 Introduction

The aim of *exact algorithms* is to exactly solve NP-hard problems in the smallest possible (exponential) worst-case running time. This field dates back to the sixties and seventies, and it has started to attract a growing interest in the last two decades. There are several explanations to the increasing interest in exact algorithms:

- There are certain applications that require exact solutions of NP-hard problems, although this might only be possible for moderate input sizes. This holds in particular for NP-complete decision problems.
- Approximation algorithms are not always satisfactory. Various problems are hard to approximate. For example, maximum independent set is hard to approximate within $O(n^{1-\varepsilon})$, for any constant $\varepsilon > 0$, unless $P = NP$ [25].
- A reduction of the base of the exponential running time, say from $O(2^n)$ to $O(1.9^n)$, increases the size of the instances solvable within a given amount of time by a constant *multiplicative* factor; running a given exponential algorithm on a faster computer can enlarge the mentioned size only by a (small) *additive* factor.
- The design and analysis of exact algorithms leads to a better understanding of NP-hard problems and initiates interesting new combinatorial and algorithmic challenges.

1.1 Branch & Reduce algorithms

One of the major techniques in the design of exact algorithms is *Branch & Reduce*, which traces back to the paper of Davis and Putnam [5] (see also [4]). A typical Branch & Reduce algorithm for a given problem $\mathcal{P}$ works as follows. If $\mathcal{P}$ is a *base instance*, the problems is solved directly in polynomial time. Otherwise the algorithm transforms the problem by applying a set of polynomial-time *reduction rules*. Then it branches, in polynomial-time, on two or more subproblems $\mathcal{P}_1, \ldots, \mathcal{P}_h$, according to a proper set of *branching rules*. Such subproblems are solved recursively, and the partial solutions obtained are eventually combined, in polynomial time, to get a solution for $\mathcal{P}$.

*Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, via del Politecnico 1, 00133 Roma, Italy, grandoni@disp.uniroma2.it.

Branch & Reduce algorithms are usually analyzed with the *bounded search tree* technique. Suppose we wish to find a time bound for a problem of size n. Assume that the depth of the search tree is polynomially bounded (which is trivially true in most cases). It is sufficient to bound the maximum number $P(n)$ of base instances generated by the algorithm: the running time will be $O^*(P(n))$[1]. If $\mathcal{P}$ is a base instance, trivially $P(n) = 1$. Otherwise, consider a possible branching step b, generating subproblems $\mathcal{P}^b_1, \ldots, \mathcal{P}^b_{h(b)}$, and let $n - \delta^b_j < n$ be the size of subproblem $\mathcal{P}^b_j$. The vector $\delta^b = (\delta^b_1, \ldots, \delta^b_{h(b)})$ is sometimes called *branching vector*. It follows that

$$P(n) \leq \sum_{j=1}^{h(b)} P(n - \delta^b_j).$$

Consider function

$$f^b(x) = 1 - \sum_{j=1}^{h(b)} x^{-\delta^b_j}.$$

This function has a unique positive root $\lambda^b = bf(\delta^b)$ (*branching factor* of δ^b). Branching factors can be easily computed numerically (see Appendix A). It turns out that $P(n) \leq \lambda^n$, where $\lambda = \max_b\{\lambda^b\}$.

We say that a branching vector δ *dominates* a branching vector δ' if $\delta \leq \delta'$, i.e. δ is component-wise not larger than δ'. It is not hard to see that, when $\delta \leq \delta'$, $bf(\delta) \geq bf(\delta')$. Hence, with respect to the running time analysis, it is sufficient to consider a dominating set of branching vectors. In other words, each time we replace the branching vector of a feasible branching with a branching vector dominating it, we obtain a pessimistic estimate of the running time. These properties will be extensively used in these notes.

1.2 Measure & Conquer

Branch & Reduce algorithms have been used for more than 40 years to solve NP-hard problems. The fastest known such algorithms are often very complicated. Typically, they consist of a long list of non-trivial branching and reduction rules, and are designed by means of a long and tedious case distinction. Despite that, the analytical tools available are still far from producing tight worst-case running time bounds for this kind of algorithms.

In these notes we present an improved analytical tool, that we called *Measure & Conquer*. In the standard analysis, n is both the measure used in the analysis and the quantity in terms of which the final time bound is expressed. However, one is free to use any, possibly sophisticated, measure m in the analysis, provided that $m \leq f(n)$ for some known function f. This way, one achieves a time bound of the kind $O^*(\lambda^m) = O^*(\lambda^{f(n)})$, which is in the desired form. The idea behind Measure & Conquer is focusing on the choice of the measure. In fact, a more sophisticated measure may capture phenomena which standard measures are not able to exploit, and hence lead to a tighter analysis of a *given* algorithm.

We apply Measure & Conquer to a toy algorithm `mis` for MIS. According to a standard analysis, the running time of this algorithm is $O^*(1.33^n)$. Thanks to a better measure, we prove that the *same* algorithm has indeed running time $O^*(1.26^n)$. This result shows that a good choice of the measure can have a tremendous impact on the time bounds achievable,

[1]Throughout this paper we use a modified big-Oh notation that suppresses all polynomially bounded factors. For functions f and g we write $f(n) = O^*(g(n))$ if $f(n) = O(g(n)poly(n))$, where $poly(n)$ is a polynomial. Also while speaking about graph problems, we use n to denote the number of nodes in the graph.

comparable to the impact of improved branching and reduction rules. Hence, finding a good measure should be at first concern when designing Branch & Reduce algorithms.

2 The Maximum Independent Set Problem

Let $G = (V, E)$ be an n-node undirected, simple graph without loops. Sometimes, we also use $V(G)$ for V and $E(G)$ for E. The (open) *neighborhood* of a node v is denoted by $N(v) = \{u \in V : uv \in E\}$, and its closed neighborhood by $N[v] = N(v) \cup \{v\}$. We let $d(v) = |N(v)|$ be the *degree* of v. By $N^x(v)$ we denote the set of nodes at distance x from v. In particular, $N^1(v) = N(v)$. Given a subset V' of nodes, $G[V']$ is the graph induced by V', and $G - V' = G[V \setminus V']$. We use $G - v$ for $G - \{v\}$.

A set $S \subseteq V$ is called an *independent set* for G if the nodes of S are pairwise non adjacent. The *independence number* $\alpha(G)$ of a graph G is the maximum cardinality of an independent set of G. The *maximum independent set* problem (`MIS`) asks to determine $\alpha(G)$.

Suppose that the considered algorithm, at a given branching or reduction step, decides that a node v belongs or does not belong to the optimum solution. In the first case we say that v is *selected*, and otherwise *discarded*.

Let us describe some simple properties of maximum independent sets.

Lemma 1 *Let G be a graph with a connected component $C \subset G$. Then*

$$\alpha(G) = \alpha(C) + \alpha(G - C).$$

Lemma 2 *Let G be a graph and v and w two nodes of G with $N[w] \subseteq N[v]$ (w* dominates *v), then*

$$\alpha(G) = \alpha(G - v).$$

Lemma 3 *Let G be a graph and v any node of G. Then there exists a maximum independent set either containing v or at least two of its neighbors $N(v)$.*

Exercise 1 *Prove Lemmas 1, 2, and 3.*

We will use the following folding operation, which is a special case of the *struction* operation defined in [6], and which was introduced in the context of exact algorithm for `MIS` in [1, 3]. A node v is *foldable* if $N(v) = \{u_1, u_2, \ldots, u_{d(v)}\}$ contains no anti-triangle[2]. *Folding* a given foldable node v of G is the process of transforming G into a new graph G_v by:

(1) adding a new node u_{ij} for each anti-edge $u_i u_j$ in $N(v)$;

(2) adding edges between each u_{ij} and the nodes in $N(u_i) \cup N(u_j) \setminus N[v]$;

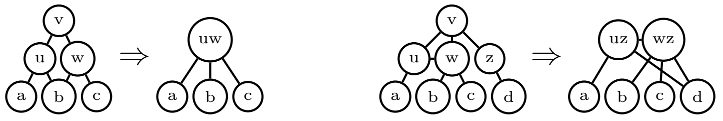
(3) adding one edge between each pair of new nodes;

(4) removing $N[v]$.

Note that nodes of degree at most two are always foldable. Examples of folding are given in Figure 1. The following simple property holds.

[2]An anti-triangle is a triple of nodes which are pairwise not adjacent. Similarly, an anti-edge is a pair of non-adjacent nodes.

Figure 1 Examples of folding.



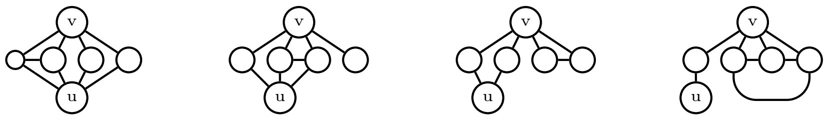
Lemma 4 *Consider a graph G, and let G_v be the graph obtained by folding a foldable node v. Then*

$$\alpha(G) = 1 + \alpha(G_v).$$

Exercise 2 *Prove Lemma 4. Hint: use Lemma 3*

We eventually introduce the following useful notion of mirror defined in [11, 13]. Given a node v, a *mirror* of v is a node $u \in N^2(v)$ such that $N(v) \setminus N(u)$ is a (possibly empty) clique. We denote by $M(v)$ the set of mirrors of v. Examples of mirrors are given in Figure 2. Intuitively, when we discard a node v, we can discard its mirrors as well without modifying the maximum independent set size. This intuition is formalized in the following lemma.

Figure 2 Example of mirrors: u is a mirror of v.



Lemma 5 *For any graph G and for any node v of G,*

$$\alpha(G) = \max\{\alpha(G - v - M(v)), 1 + \alpha(G - N[v])\}.$$

Exercise 3 *Prove Lemma 5. Hint: use Lemma 3*

3 A Simple MIS Algorithm

In this section we describe a toy algorithm `mis` for MIS, and show that its running time is $O^*(1.33^n)$ via a standard analysis. Algorithm `mis` is described in Figure 3. When the graph is empty (base case) the algorithm simply returns 0. Otherwise it applies, when possible, the Folding Lemma 4 to a node v of degree at most 2, considering nodes of smaller degree first in case of ties:

$$\texttt{mis}(G) = 1 + \texttt{mis}(G_v).$$

As a last choice, the algorithm *greedily* takes a node v of maximum degree, and branches by either discarding v or selecting v (and discarding its neighbors)

$$\texttt{mis}(G) = max\{\texttt{mis}(G - v), 1 + \texttt{mis}(G - N[v])\}.$$

Theorem 1 *Algorithm* `mis` *solves MIS in $O^*(1.33^n)$ time.*

Figure 3 Algorithm mis for the maximum independent set problem.

```
int mis(G) {
    if ($G = \emptyset$) return 0; //Base case
    //Folding
    Take $v$ of minimum degree
    if ($d(v) \leq 2$) return 1 + mis($G_v$);
    //"Greedy" branching
    Take $v$ of maximum degree;
    return max{ mis($G - v$), 1 + mis($G - N[v]$) };
}
```

Proof. Let $P(n)$ be the number of base instances generated by the algorithm to solve an instance of size n. The depth of the recursion is $O(n)$ since in each step the number of nodes decreases at least by one. Moreover, the algorithm takes polynomial time, excluding the time needed for the recursive calls. It follows that the running time of the algorithm is $O(P(n)n^{O(1)}) = O^*(P(n))$.

We next show by induction that $P(n) \leq \lambda^n$ for $\lambda < 1.33$. In the base case $n = 0$ and $P(0) = 1 \leq \lambda^0$ for every $\lambda > 0$. When the algorithm folds a node, the number of nodes decreases at least by one. Hence, for every $\lambda \geq 1$,

$$P(n) \leq P(n-1) \leq \lambda^{n-1} \leq \lambda^n.$$

When the algorithm branches at a node v with $d(v) \geq 4$, in one subproblem it removes 1 node (i.e. v), and in the other it removes $1 + d(v) \geq 5$ nodes (i.e. $N[v]$). Let $\lambda_1 = bf(1,5) = 1.32\ldots < 1.33$ be the positive root of $1 - x^{-1} - x^{-5}$. For $\lambda \geq \lambda_1$ we obtain

$$P(n) \leq P(n-1) + P(n-5) \leq \lambda^{n-1} + \lambda^{n-5} \leq \lambda^n.$$

Otherwise, the algorithm branches at a node v of degree exactly 3, hence removing either 1 or 4 nodes. However, in the first case a node of degree 2 is folded afterwards, with the removal of at least 2 more nodes. Let $\lambda_2 = bf(3,4) = 1.22\ldots < 1.23$ be the positive root of $1 - x^{-3} - x^{-4}$. For $\lambda \geq \lambda_2$,

$$P(n) \leq P(n-3) + P(n-4) \leq \lambda^{n-3} + \lambda^{n-4} \leq \lambda^n.$$

The claim follows. □

The time bound above is the best one can achieve via a standard analysis. We will see how a non-standard analysis can provide much better time bounds.

4 A Refined Analysis via Measure & Conquer

The classical approach to *improve* on mis would be designing refined branching and reduction rules. In particular, one tries to improve on the *tight* recurrences. We next show how to get a much better time bound thanks to a better measure of subproblems size (without changing

Figure 4 Case analysis of folding for $m = n_{\geq 3}$.

$d(u)$	$d(w)$	$d(uw)$	m'
2	2	2	m
2	≥ 3	≥ 3	$m - 1 + 1$
≥ 3	≥ 3	≥ 4	$m - 2 + 1$

the algorithm!). We will start by introducing in Section 4.1 an alternative, simple, measure. This measure does not immediately give a better time bound, but it will be a good starting point to define a really better measure.

4.1 An Alternative Measure

Nodes of degree at most 2 can be removed without branching. Hence they do not really contribute to the *size* of the problem. For example, if the maximum degree is 2, then `mis` solves the problem in polynomial time! In view of that, let us define the size of the problem to be number of nodes of degree at least 3.

More formally, let n_i denote the number of nodes of degree i, and $n_{\geq i} = \sum_{j \geq i} n_j$. We define the size of the problem to be $m = n_{\geq 3}$ (rather than $m = n$). We remark that, since $m = n_{\geq 3} \leq n$, if we prove a running time bound of type $O^*(\lambda^m)$, we immediately get a $O^*(\lambda^n)$ time bound.

Let us give an alternative proof of Theorem 1.

Proof. (Theorem 1) Let us define G a base instance if the maximum degree in G is 2 (which implies $m = n_{\geq 3} = 0$). Let moreover $P(m)$ be the number of base instances generated by the algorithm to solve an instance of size m. By the usual argument the running time is $O^*(P(m))$. We prove by induction that $P(m) \leq \lambda^m$ for $\lambda < 1.33$, which implies the claim being $m \leq n$. In the base case $m = 0$. Thus

$$P(0) = 1 \leq \lambda^0.$$

Let m' be the size of the problem after folding a node v. It is sufficient to show that $m' \leq m$, from which

$$P(m) \leq P(m') \leq \lambda^{m'} \leq \lambda^m$$

for $\lambda \geq 1$. This condition trivially holds when folding only removes nodes. In the remaining case, $N(v) = \{u, w\}$ with $uw \notin E$. In this case we remove $\{v, u, w\}$ and add a node uw with $d(uw) \leq d(u) + d(w) - 2$. By case analysis (see Figure 4) $m' \leq m$ also in this case.

Suppose now that we branch at a node v with $d(v) \geq 4$. Note that all the nodes of the graph have degree ≥ 3 (since we do not fold). Hence by the standard argument

$$P(m) \leq P(m-1) + P(m-5) \leq \lambda^{m-1} + \lambda^{m-5} \leq \lambda^m.$$

Recall that the inequality above is satisfied for $\lambda \geq 1.33$.

Eventually, consider branching at v, $d(v) = 3$. In this case we remove either 1 or 4 nodes of degree 3. However, in the first case the degree of the 3 neighbors of v drops from 3 to 2, with a consequent further reduction of the size by 3:

$$P(m) \leq P(m-4) + P(m-4) \leq \lambda^{m-4} + \lambda^{m-4} \leq \lambda^m.$$

The inequality above is satisfied for $\lambda \geq bf(4,4) = 1.18\ldots$. The claim follows. $\square$

4.2 A Better Measure

When we branch at a node of large degree, we decrement by 1 the degree of many other nodes. This is beneficial on long term, since we can remove nodes of degree at most 2 without branching. We are not exploiting this fact to its full extent in the current analysis.

An idea is then to attribute a larger *weight* $\omega_i \leq 1$ to nodes v of larger degree i, and let the size of the problem be the sum of node sizes. This way, when the degree of a node decreases, the size of the problem decreases as well. More formally, for a constant $\omega \in (0,1]$ to be fixed later, we let

$$\omega_i = \begin{cases} 0 & \text{if } i \leq 2; \\ \omega & \text{if } i = 3; \\ 1 & \text{otherwise.} \end{cases}$$

We also use $\omega(v)$ for $\omega_{d(v)}$. The size $m = m(G)$ of graph $G = (V,E)$ is

$$m = \sum_{v \in V} w(v) = \omega \cdot n_3 + n_{\geq 4}.$$

Thanks to this new measure of subproblems size, we are able to refine the analysis of `mis`.

Theorem 2 *Algorithm* `mis` *solves MIS in* $O^*(1.29^n)$ *time.*

Proof. With the usual notation, let us show that $P(m) \leq \lambda^m$ for $\lambda < 1.29$. In the base case $m = 0$, and thus

$$P(0) = 1 \leq \lambda^0.$$

In case of folding of node v, let $m' = m(G_v)$ be the size of the corresponding subproblem. Also in this case it is sufficient to show that $m' \leq m$. This condition is satisfied when nodes are only removed (being the weight increasing with the degree). The unique remaining case is $N(v) = \{u, w\}$, with u and w not adjacent. In this case we remove $\{v, u, w\}$, and add a node uw with $d(uw) \leq d(u) + d(w) - 2$. Hence it is sufficient to show that

$$\omega(v) + \omega(u) + \omega(w) - \omega(uw) = \omega(u) + \omega(w) - \omega(uw) \geq 0.$$

By a simple case analysis (see Figure 5), it follows that this condition holds for $\omega \geq 0.5$.

Consider now the case of branching at a node v, $d(v) \geq 5$. Let d_i be the degree of the ith neighbor of v (which thus has weight ω_{d_i}). Then

$$\begin{aligned} P(m) &\leq P(m - \omega_{d(v)} - \sum_{i=1}^{d(v)} (\omega_{d_i} - \omega_{d_i - 1})) + P(m - \omega_{d(v)} - \sum_{i=1}^{d(v)} \omega_{d_i}) \\ &\leq P(m - 1 - \sum_{i=1}^{5} (\omega_{d_i} - \omega_{d_i - 1})) + P(s - 1 - \sum_{i=1}^{5} \omega_{d_i}). \end{aligned}$$

Observe that we can replace $d_i \geq 6$ with $d_i = 5$. In fact in both cases $\omega_{d_i} = 1$ and $\omega_{d_i} - \omega_{d_i - 1} = 0$. Hence we can assume $d_i \in \{3,4,5\}$. This is crucial to obtain a finite number of recurrences! We obtain the following set of recurrences

$$P(m) \leq P(m - 1 - t_3(\omega - 0) - t_4(1 - \omega) - t_5(1 - 1)) + P(m - 1 - t_3\,\omega - t_4 - t_5),$$

Figure 5 Case analysis of folding for $m = \omega n_3 + n_{\geq 4}$.

$d(u)$	$d(w)$	$d(uw)$	$\omega(u) + \omega(w) - \omega(uw) \geq 0$
2	2	2	$0 + 0 - 0 \geq 0$
2	3	3	$0 + \omega - \omega \geq 0$
2	≥ 4	≥ 4	$0 + 1 - 1 \geq 0$
3	3	4	$\omega + \omega - 1 \geq 0$
3	≥ 4	≥ 4	$\omega + 1 - 1 \geq 0$
≥ 4	≥ 4	≥ 4	$1 + 1 - 1 \geq 0$

where t_3, t_4, and t_5 are non-negative integers satisfying $t_3 + t_4 + t_5 = 5$. (Intuitively, t_i is the number of neighbors of v of degree i).

Consider now branching at a node v, $d(v) = 4$. By a similar argument (with $d_i \in \{3, 4\}$), we obtain

$$P(m) \leq \begin{cases} P(m - 1 - 4 \cdot \omega - 0 \cdot (1 - \omega)) + P(m - 1 - 4 \cdot \omega - 0 \cdot 1) \\ P(m - 1 - 3 \cdot \omega - 1 \cdot (1 - \omega)) + P(m - 1 - 3 \cdot \omega - 1 \cdot 1) \\ P(m - 1 - 2 \cdot \omega - 2 \cdot (1 - \omega)) + P(m - 1 - 2 \cdot \omega - 2 \cdot 1) \\ P(m - 1 - 1 \cdot \omega - 3 \cdot (1 - \omega)) + P(m - 1 - 1 \cdot \omega - 3 \cdot 1) \\ P(m - 1 - 0 \cdot \omega - 4 \cdot (1 - \omega)) + P(m - 1 - 0 \cdot \omega - 4 \cdot 1) \end{cases}$$

Consider eventually branching at a node v, $d(v) = 3$. By an analogous argument (with $\omega(v) = \omega_3 = \omega$ and $d_i = 3$)

$$P(m) \leq P(m - \omega - 3\,\omega) + P(m - \omega - 3\,\omega).$$

For every $\omega \in [1/2, 1]$, the set of recurrences above provides an upper bound $\lambda(\omega)$ on λ. Our goal is minimizing $\lambda(\omega)$ (hence getting a better time bound). Via exhaustive search over a grid of values for ω we obtained $\lambda(0.7) < 1.29$ (see Appendix B for a C++ program computing it). The claim follows. $\square$

4.3 An Even Better Measure

We can extend the analysis from previous section to larger degrees. For example, we might let the weight ω_i associated to degree-i nodes be:

$$\omega_i = \begin{cases} 0 & \text{if } i \leq 2; \\ \omega & \text{if } i = 3; \\ \omega' & \text{if } i = 4; \\ 1 & \text{otherwise.} \end{cases}$$

Here ω and ω' are two proper constants, with $0 < \omega \leq \omega' \leq 1$. Using this measure, and an analysis similar to the one from previous section, it is not hard to prove the following result (see Appendix C for a C++ program optimizing the weights).

Theorem 3 *Algorithm* `mis` *solves MIS in* $O^*(1.26^n)$ *time.*

Exercise 4 *Prove the theorem above. Hint:* $\omega = 0.750$ *and* $\omega' = 0.951$*.*

Exercise 5 *What happens if we set* $\omega_5 = \omega''$ *for a further weight* $\omega' \leq \omega'' \leq 1$*? Do you see any pattern?*

Exercise 6 *Design a better algorithm for MIS, possibly using the other mentioned reduction rules (mirroring etc.). Analyze this algorithm in the standard way and via Measure & Conquer.*

Exercise 7 *Can you see an alternative, promising measure for MIS?*

5 Lower bounds

Despite the big improvements in the running time bounds, it might be that our refined analysis is still far from being tight. Hence, it is natural to ask for (exponential) lower bounds. Notice that we are concerned with lower bounds on the complexity of a particular algorithm, and not with lower bounds on the complexity of an algorithmic problem. A lower bound may give an idea of how far the analysis is from being tight.

In this section we prove a $\Omega(2^{n/4})$ lower bound on the running time of `mis`. The large gap between the upper and lower bound for `mis` suggests the possibility that the analysis of that algorithm can be further refined (possibly by measuring the size of the subproblems in a further refined way).

Theorem 4 *The running time of* `mis` *is* $\Omega(2^{n/4}) = \Omega(1.18^n)$*.*

Proof. Consider the graph G_k consisting of $k = n/4$ copies of a 4-clique (see Figure 6). We let $P(k)$ be the number of subproblems generated by `mis` to solve MIS on G_k. Consider any clique $C = \{a, b, c, d\} \in G_k$. The algorithm might branch at a. In the subproblem where a is discarded, the algorithm removes b, c, and d via folding. In the other subproblem the algorithm removes $N[a] = \{a, b, c, d\}$. Hence in both cases C is deleted from the graph, leaving an instance G_{k-1} which is solved recursively. We thus obtain the following recurrences:

$$P(k) \geq \begin{cases} 2P(k-1) & \text{if } k \geq 1; \\ 1 & \text{if } k = 0. \end{cases}$$

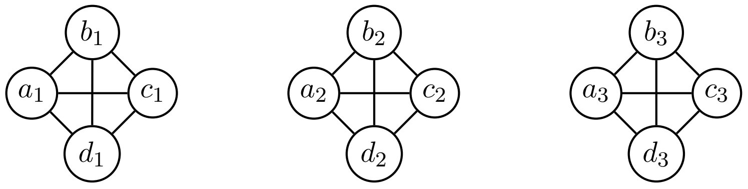
We can conclude that $P(k) \geq 2^k = 2^{n/4}$. □

Exercise 8 *Find a larger lower bound on the running time of* `mis`*. Hint:* $\Omega(3^{n/6}) = \Omega(1.20^n)$*, maybe better.*

Exercise 9 *Consider the variant of* `mis` *where the algorithm, after the base case, branches on connected components when possible. Can you find a good lower bound on the running time of this modified algorithm?*

Remark 1 *Typically finding lower bounds on connected graphs is much more complicated.*

Figure 6 Example of the lower bound graph G_k for $k = 3$.



6 Quasiconvex Analysis of Backtracking Algorithms

When the number of distinct weights grows, there is a computational problem which one has to face. In fact, both the number of recurrences and the space of candidate weights tend to grow exponentially in the number of weights. Of course the best weights need to be computed only once, and this computation has no impact on the actual behavior of the algorithm. Still, this can be a problem during the design of the algorithm, when having a quick feedback is important. In this section we outline a general way to cope with the optimization of the weights (for a given set of recurrences), described by Eppstein [8].

6.1 Multivariate Recurrences

Consider a collection of integral *measures* $m_1, \ldots, m_d$, describing different aspects of the size of the problem considered. For example, in the analysis of `mis` in Section 4.2 we used $m_1 = n_3$ and $m_2 = n_{\geq 4}$. Let $P(m_1, \ldots, m_d)$ be the number of base instances generated by the algorithm to solve a problem with measures $m_1, \ldots, m_d$. Consider a given branching step b, and let $\delta^b_{i,j}$ be the decrease of the ith measure of the jth subproblem. The following multivariate recurrence holds

$$\begin{aligned} P(m_1, \ldots, m_d) \leq P(m_1 - \delta^b_{1,1}, \ldots, m_d - \delta^b_{d,1}) + \ldots \\ + P(m_1 - \delta^b_{1,h(b)}, \ldots, m_d - \delta^b_{d,h(b)}) \end{aligned}$$

Remark 2 *Some of the $\delta^b_{i,j}$'s might be negative. For example, when we delete one edge incident to a node of degree 4, $n_{\geq 4}$ decreases but n_3 grows.*

Solving multivariate recurrences is typically rather complicated. A common alternative is turning them into univariate recurrences by considering a linear combination of the measures (*aggregated measure*)

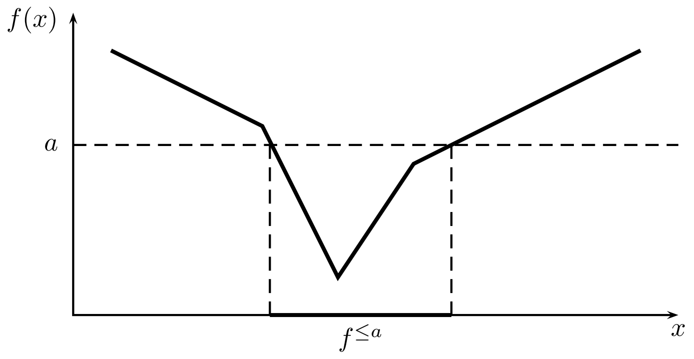
$$m(w) = w_1 \, m_1 + \ldots + w_d \, m_d$$

Here $w = (w_1, \ldots, w_d)$ plays a role analogous to the weights ω_i in the analysis of `mis`. In particular, in Section 4.2 we set $w_1 = \omega \in (0, 1]$ and $w_2 = 1$.

The weights w_i are in general rational, possibly negative, numbers. However, they need to satisfy the constraint $\delta^b_j := \sum_i w_i \, \delta^b_{i,j} > 0$ for every branching b and corresponding subproblem j. In words, the aggregated measure $m(w)$ decreases in each subproblem[3]. For example, in the analysis of `mis`, this condition is satisfied for every $\omega \in [0.5, 1]$.

[3]In the degenerate case $h(b) = 1$ (a unique subproblem), $\delta^b_1 = \delta^b_{1,1} \geq 0$ is allowed (provided that the branching depth can be bounded in an alternative way).

Figure 7 Example of quasiconvex function, with a corresponding convex level set.



The resulting set of univariate recurrences can be solved in the standard way (for fixed weights). In particular, for each branching b, we compute the (unique) positive root $\lambda^b(w)$ of function

$$f^b(x,w) := 1 - \sum_{j=1}^{h(b)} x^{-\sum_{i=1}^{d} w_i \delta^b_{i,j}}.$$

This gives a running time bound of the kind $O^*(\lambda(w)^{\sum_i w_i m_i})$ where $\lambda(w) := \max_b\{\lambda^b(w)\}$.

6.2 Quasiconvexity

Function $\lambda(w)$ has a very special property, which simplifies considerably its minimization. We recall that a function $f : D \to \mathbb{R}$, with $D \subseteq \mathbb{R}^d$ convex, is *quasiconvex* if its level set

$$f^{\leq a} := \{x \in D : f(x) \leq a\}$$

is convex for any $a \in \mathbb{R}$. An example of quasiconvex (but not convex) function is given in Figure 7.

Theorem 5 *Function $\lambda(w)$ is quasiconvex over $\mathbb{R}^d$.*

Proof. The maximum over a finite number of quasiconvex functions is quasiconvex. Hence it is sufficient to show that each $\lambda^b(w)$ is quasiconvex. Recall that $\lambda^b(w)$ is the unique positive root of $f^b(x,w) = 1 - \sum_j x^{-\sum_i w_i \delta^b_{i,j}}$. Define $g^b(x,w) = \sum_j x^{-\sum_i w_i \delta^b_{i,j}}$. From the monotonicity of $f^b(x,w)$

$$\lambda^{b,\leq a} = \{w \in \mathbb{R}^d : f^b(x,w) \geq 0\} = \{w \in \mathbb{R}^d : g^b(x,w) \leq 1\} = g^{b,\leq 1}.$$

Function g^b is the sum of convex functions, and hence is convex. Then trivially its level sets, including $g^{b,\leq 1}$, are convex. □

Corollary 1 *Function $\lambda(w)$ is quasiconvex over any convex $D \subseteq \mathbb{R}^d$.*

Proof. It follows from the proof of Theorem 5, and the fact that the intersection of convex sets is convex. □

6.3 Applications to Measure & Conquer

We can use Theorem 5 to optimize the weights in a much faster way with respect to exhaustive grid search. Suppose we define a set of linear constraints on the weights such that:

(a) the size of each subproblem does not increase;

(b) $m(w) \leq n$, where n is a *standard* measure for the problem.

This gives a convex domain of weights w. On that domain we can compute the minimum value $\lambda(\tilde{w})$ of the quasiconvex function $\lambda(w)$. The resulting running time is $O^*(\lambda(\tilde{w})^n)$.

There are known techniques to find efficiently the minimum of a quasi-convex function (see e.g. [8]). We successfully applied [12, 13, 14] the following, very fast and easy to implement, approach based on *randomized local search* (in simulated annealing style):

- We start from any feasible initial value w;
- We add to w a random vector $\Delta\, w$ in a given range $[-\Delta, \Delta]^d$;
- If the resulting w' is feasible and gives $\lambda(w') \leq \lambda(w)$, we set $w = w'$;
- We iterate the process, reducing the value of Δ if no improvement is achieved for a *large* number of steps;
- The process halts when Δ drops below a given value Δ'.

Appendix D contains a C++ program applying this method to the optimization of the weights in the analysis of Section 4.3.

Remark 3 *The local search algorithm above does not guarantee closeness to the optimal $\lambda(\tilde{w})$. However it is accurate in practice. More important, it always provides* feasible *upper bounds on the running time.*

7 Other Examples of Measure & Conquer

In this section we briefly describe other (more or less explicit) applications of the Measure & Conquer approach.

The first non-trivial algorithm for the minimum dominating set problem (MDS) is based on Measure & Conquer [17, 18][4]. The basic idea is developing an algorithm for the minimum set cover problem (MSC). This algorithm is analyzed by measuring the size of the subproblems in terms of the sum of the number n of sets and number m of elements. The resulting running time is $O^*(1.381^{n+m})$. The size of the MSC formulation of a MDS instance on n nodes is $2n$. It follows that MDS can be solved in $O^*(1.381^{2n}) = O^*(1.803^n)$ time. The same algorithm is re-analyzed in [10, 13], using a refined measure which assigns different weights to sets of different size and elements of different frequency. This way the time bound is refined to $O^*(1.527^n)$, an impressive improvement.

A similar, but more complex measure is used in [12] to develop the first better-than-trivial algorithm for the connected version of MDS, where the dominating set is required to induce a connected graph. Here, besides cardinalities and frequencies, the measure takes into account the local connectivity properties of the original graph.

[4]In the same year, slower but better-than-trivial algorithms for MDS were independently developed in [15, 22].

In a paper on 3-coloring and related problems [2], Beigel and Eppstein consider a reduction to constraint satisfaction, and measure the size of the constraint satisfaction problem with a linear combination of the number of variables with three and four values in their domain, respectively. A more sophisticated measure is introduced by Eppstein in the context of cubic-TSP [7]: let F be a given set of *forced* edges, that is edges that we assume belonging to the optimum solution. For an input cubic graph $G = (V, E)$, the author measures the size of the problem in terms of $|V| - |F| - |C|$, where C is the set of 4-cycles which form connected components of $G - F$.

Gupta et al. [19] used Measure & Conquer while analyzing exact algorithms for finding maximal induced subgraphs of fixed node degree. Razgon [23], using a non-standard measure, derived the first non-trivial algorithm breaking the $O^*(2^n)$ barrier for the feedback vertex set problem (see also [9]). Kowalik [20] used Measure & Conquer in his branching algorithm for the edge coloring problem. The analysis of Gasper-Liedloff's algorithm for the independent dominating set problem in [16] is based on Measure & Conquer. Another example is the paper by Kratsch and Liedloff on the minimum dominating clique problem [21]. We are also aware of a number of other (still unpublished) papers using the same kind of approach.

Measure & Conquer can be used also as a tool to prove tighter combinatorial bounds. For example, using this kind of approach and the same measure which is mentioned above for `MDS`, Fomin et al. [14] proved that the number of minimal dominating sets in a graph is $O^*(1.721^n)$. Based on this result, they also derived the first non-trivial exact algorithms for the domatic number problem and for the minimum-weight dominating set problem. The bounds on the number of minimal feedback vertex sets (or maximal induced forests) obtained in [9] are also based on Measure & Conquer.

Of course, a non-standard measure can be used to design better algorithms in the standard way: one considers the tight recurrences for a given algorithm (and measure), and tries to design better branching and reduction rules for the corresponding cases. A very recent work by van Rooij and Bodlaender goes in this direction [24].

References

[1] R. Beigel. Finding maximum independent sets in sparse and general graphs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 856–857, 1999.

[2] R. Beigel and D. Eppstein. 3-coloring in time O(1.3289^n). *Journal of Algorithms*, 54(2):168–204, 2005.

[3] J. Chen, I. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5:394–397, 1962.

[5] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[6] C. Ebenegger, P. L. Hammer, and D. de Werra. Pseudo-boolean functions and stability of graphs. *Annals of Discrete Mathematics*, 19:83–98, 1984.

[7] D. Eppstein. The Traveling Salesman Problem for Cubic Graphs. *Journal of Graph Algorithms and Applications*, 11(1):61–81, 2007.

[8] D. Eppstein. Quasiconvex analysis of backtracking algorithms. *ACM Transactions on Algorithms*, 2(4):492–509, 2006.

[9] F. V. Fomin, S. Gaspers, A. V. Pyatkin, and I. Razgon. On the Minimum Feedback Vertex Set Problem: Exact and Enumeration Algorithms. *Algorithmica*, 52(2):293–307, 2008.

[10] F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: domination - a case study. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 191–203, 2005.

[11] F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: a simple $O(2^{0.288\,n})$ independent set algorithm. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 18–25, 2006.

[12] F. V. Fomin, F. Grandoni, and D. Kratsch. Solving connected dominating set faster than 2^n. *Algorithmica,* 52(2):153–166, 2008.

[13] F. V. Fomin, F. Grandoni, and D. Kratsch. A Measure & Conquer Approach for the Analysis of Exact Algorithms. To appear in *Journal of the ACM.*

[14] F. V. Fomin, F. Grandoni, A. Pyatkin, and A. Stepanov. Combinatorial bounds via measure and conquer: Bounding minimal dominating sets and applications. *ACM Transactions on Algorithms,* 5(1): 2008.

[15] F. V. Fomin, D. Kratsch, and G. J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 199–210, 2004.

[16] S. Gaspers and M. Liedloff. A branch-and-reduce algorithm for finding a minimum independent dominating set in graphs. In *Graph-Theoretic Concepts in Computer Science (WG)*, pages 78–89, 2006.

[17] F. Grandoni. *Exact Algorithms for Hard Graph Problems.* PhD thesis, Università di Roma "Tor Vergata", Roma, Italy, Mar. 2004.

[18] F. Grandoni. A note on the complexity of minimum dominating set. *Journal of Discrete Algorithms*, 4(2):209–214, 2006.

[19] S. Gupta, V. Raman, and S. Saurabh. Fast exponential algorithms for maximum -regular induced subgraph problems. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 139–151. 2006.

[20] L. Kowalik. Improved edge-coloring with three colors. In *Graph-Theoretic Concepts in Computer Science (WG)*, pages 90–101. 2006.

[21] D. Kratsch and M. Liedloff. An exact algorithm for the minimum dominating clique problem. *Theoretical Computer Science,* 385(1-3):226–240, 2007.

[22] B. Randerath and I. Schiermeyer. Exact algorithms for MINIMUM DOMINATING SET. Technical Report zaik-469, Zentrum für Angewandte Informatik, Köln, Germany, 2004.

[23] I. Razgon. Exact computation of maximum induced forest. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 160–171, 2006.

[24] J. van Rooij and H. L. Bodlaender. Design by Measure and Conquer, A Faster Exact Algorithm for Dominating Set. In *Symposium on Theoretical Aspects of Computer Science,* pages 657–668, 2008.

[25] D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In *ACM Symposium on the Theory of Computing (STOC)*, pages 681–690, 2006.

Appendix A

```
#include <cstdlib>
#include <iostream>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

#define PRECISION 40
#define MAXN 100

using namespace std;

/*
branchFactor() receives in input a branching vector
It returns the corresponding branching factor "bf"
This is done via doubling + binary search. The desired value satisfies
1-sum_{j=0}^{n-1} bf^(-V[i])=0
The binary search is anyway interrupted when the value is accurate enough
*/
double branchFactor(int n, double* V){

  double left=0.0;
  double right=1.0;
  double f;
  //we compute an upper bound right on the branching factor via doubling
  do{
    right = right*2;
    f=1.0;
    for(int j=0; j<n; j++) {
      f -= pow(right, -V[j]);
    }
  }while(f<=0);
  //we compute the branching factor via binary search
  double bf;
  for(int i=0; i<PRECISION; i++) {
    bf = (left+right)/2.0;
    f=1.0;
    for(int j=0; j<n; j++) {
      f -= pow(bf, -V[j]);
    }
    if(f==0) return bf;
    if(f>0) right = bf;
    else left = bf;
  }
  return right;//this way we return an upper bound
}


int main() {

  int n;
  double V[MAXN];
  double d;

  while(1) {
    cout << endl << "# of branchings:";
    cin >> n;
    if(n<=0 || n>MAXN) exit(1);

    for(int i=0; i<n; i++) {
      cout << "delta(" << i << "):";
      cin >> d;
      if(d<0) exit(1);
      V[i]=d;
    }

    double bf = branchFactor(n, V);
    printf("bf=%.8f (%.8f)\n", bf, log(bf)/log(2));
  }
  return 0;
}
```

Appendix B

```
#include <stdlib.h>
...

using namespace std;

double branchFactor(int n, double* V){
  ...
}
```

```
/*
lambda is the value of the branching factor for given (feasible) weights.
"stampa" is used to print or not some details
*/
double computeLambda(double a, int stampa) {


  double V[10];//we assume that we branch on at most 10 subproblem, which is the case her
  double lambda, lambda_max= -1;

  //We don't need to consider the base case and folding

  //Branch at degree >=5
  //n3 and n4 are the neighbors of degree 3 and 4
  //n5 are the neighbors of degree at least 5
  for(int n3=0; n3<=5; n3++){
    for(int n4=0; n4<=5-n3; n4++){
      int n5 = 5-n3-n4;
      V[0]=1+n3*a+n4*(1-a)+n5*0;
      V[1]=1+n3*a+n4*1+n5*1;
      lambda = branchFactor(2, V);
      if(stampa) printf("branch␣at␣5:␣1+%d*a+%d*(1-a)+%d*0/1+%d*a+%d*1+%d*1=%.8lf\n", n3,
      lambda_max = MAX(lambda_max, lambda);
    }
  }

  //Branch at degree 4
  for(int n3=0; n3<=4; n3++){
    int n4 = 4-n3;
    V[0]=1+n3*a+n4*(1-a);
    V[1]=1+n3*a+n4*1;
    lambda = branchFactor(2, V);
    if(stampa) printf("branch␣at␣4:␣1+%d*a+%d*(1-a)/1+%d*a+%d*1=%.8lf\n", n3, n4, n3, n4,
    lambda_max = MAX(lambda_max, lambda);
  }


  //Branch at degree 3
  V[0]=4*a;
  V[1]=4*a;
  lambda = branchFactor(2, V);
  if(stampa) printf("branch␣at␣3:␣4a/4a=%.8lf\n", lambda);
  lambda_max = MAX(lambda_max, lambda);


  if(stampa) printf("\nlambda_max(a=%.8lf)=%.8lf\n", a, lambda_max);

  return lambda_max;
}
```

```
//Here we put the constraints on the weights: in our case a\in [0.5,1.0]
bool feasible(double a){
  if(a>=0.5 && a<=1.0) return true;
  else return false;
}

int main() {

  double lambdamin = 1000;
  double amin = 1000;

  //we search for the best a in a grid with offset 1/grid
  int grid=100;
  for(int i=1; i<=grid; i++){
    double a=(double)i/grid;
    if(feasible(a)){
      double lambda = computeLambda(a,0);
      printf("lambda(a=%lf)=%.8lf\n", a, lambda);
      if(lambda < lambdamin){
        lambdamin = lambda;
        amin = a;
      }
    }
  }
  printf("\n\nBEST_BOUND\n");
  computeLambda(amin,1);
//   printf("\nlambdamin(amin=%lf)=%.8lf\n", amin, lambdamin);


  system("PAUSE");
  return 0;

}
```

Appendix C

```
#include <stdlib.h>
...

using namespace std;

double branchFactor(int n, double* V){
  ...
}



/*
```

```
lambda is the value of the branching factor for given (feasible) weights.
"stampa" is used to print or not some details
*/
double computeLambda(double a3, double a4, int stampa) {


  double V[10];//we assume that we branch on at most 10 subproblem, which is the case her
  double lambda, lambda_max= -1;

  //We don't need to consider the base case and folding

  //Branch at degree >=6
  //Here n6 are the neighbors of degree at least 6
  for(int n3=0; n3<=6; n3++){
    for(int n4=0; n4<=6-n3; n4++){
      for(int n5=0; n5<=6-n3-n4; n5++){
        int n6 = 6-n3-n4-n5;
        V[0]=1+n3*a3+n4*(a4-a3)+n5*(1-a4)+n6*0;
        V[1]=1+n3*a3+n4*a4+n5*1+n6*1;
        lambda = branchFactor(2, V);
        if(stampa) printf("branch_at_6:_1+%d*a3+%d*(a4-a3)+%d*(1-a4)+%d*0/1+%d*a3+%d*a4+%
                          n3, n4, n5, n6, n3, n4, n5, n6, lambda);
        lambda_max = MAX(lambda_max, lambda);
      }
    }
  }

  //Branch at degree 5
  for(int n3=0; n3<=5; n3++){
    for(int n4=0; n4<=5-n3; n4++){
      int n5 = 5-n3-n4;
      V[0]=1+n3*a3+n4*(a4-a3)+n5*(1-a4);
      V[1]=1+n3*a3+n4*a4+n5*1;
      lambda = branchFactor(2, V);
      if(stampa) printf("branch_at_5:_1+%d*a3+%d*(a4-a3)+%d*(1-a4)/1+%d*a3+%d*a4+%d*1=%.8
                          n3, n4, n5, n3, n4, n5, lambda);
      lambda_max = MAX(lambda_max, lambda);
    }
  }

  //Branch at degree 4
  for(int n3=0; n3<=4; n3++){
    int n4 = 4-n3;
    V[0]=a4+n3*a3+n4*(a4-a3);
    V[1]=a4+n3*a3+n4*a4;
    lambda = branchFactor(2, V);
    if(stampa) printf("branch_at_4:_a4+%d*a3+%d*(a4-a3)/a4+%d*a3+%d*a4=%.8lf\n",
                          n3, n4, n3, n4, lambda);
    lambda_max = MAX(lambda_max, lambda);
  }

  //Branch at degree 3
  int n3 = 3;
  V[0]=a3+n3*a3;
  V[1]=a3+n3*a3;
  lambda = branchFactor(2, V);
  if(stampa) printf("branch_at_3: a3+%d*a3/a3+%d*a3=%.8lf\n",
                          n3, n3,lambda);
  lambda_max = MAX(lambda_max, lambda);


  if(stampa) printf("\nlambda_max(a3=%.8lf,a4=%.8lf)=%.8lf\n", a3, a4, lambda_max);

  return lambda_max;
}



//Here we put the constraints on the weights
bool feasible(double a3, double a4){
  if(a4<=1 && a3<=a4 && a3>0 && 2*a3>=a4 && 2*a4>=1 && a4>=1-a3) return true;
  else return false;
}

int main() {

  double lambda_min = 1000;
  double a3min = 1000;
  double a4min = 1000;

  //we search for the best a in a grid with offset 1/grid
  int grid=100;
  for(int i=1; i<=grid; i++){
    for(int j=1; j<=grid; j++){
      double a3=(double)i/grid;
      double a4=(double)j/grid;
      if(feasible(a3,a4)){
        double lambda = computeLambda(a3,a4,0);
        printf("lambda(a3=%lf,a4=%lf)=%.8lf\n", a3, a4, lambda);
        if(lambda < lambda_min){
          lambda_min = lambda;
          a3min = a3;
          a4min = a4;
        }
      }
    }
  }
  printf("\n\nBEST_BOUND\n");
  computeLambda(a3min,a4min,1);

  system("PAUSE");
  return 0;

}
```

Appendix D

```
#include <stdlib.h>
...

using namespace std;


double branchFactor(int n, double* V){
  ...
}

double uniform(double left, double right) {
  return left + (right-left)*((double)rand())/RAND_MAX;
}


double computeLambda(double a3, double a4, int stampa) {
  ...
}


bool feasible(double a3, double a4){
  ...
}

int main() {

  double lambda_min;
  double a3min;
  double a4min;

  //compute an initial random solution
  do{
    a3min = uniform(0.0,1.0);
    a4min = uniform(0.0,1.0);
  }while(!feasible(a3min,a4min));
  lambda_min = computeLambda(a3min,a4min,0);
  printf("lambda(a3=%lf,a4=%lf)=%.8lf\n", a3min, a4min, lambda_min);


  double deltamin = 0.0001; //final step
  double delta = 0.01; //initial step
  int counter = 0; //measures for how long we don't make any progress
  do{
    double a3rand = a3min + delta*uniform(-1.0,1.0);
    double a4rand = a4min + delta*uniform(-1.0,1.0);
    counter++; //a new proposal is generated
    if(feasible(a3rand,a4rand)) {
      double lambda_rand = computeLambda(a3rand,a4rand,0);
      if(lambda_rand < lambda_min) {
        counter = 0;
```

```
          lambda_min = lambda_rand;
          a3min = a3rand;
          a4min = a4rand;
          printf("lambda(a3=%lf,a4=%lf)=%.8lf\n", a3min, a4min, lambda_min);
        }
      }
      if(counter >= 1000){
        counter = 0;
        delta = delta/2.0;
        printf("\ndelta=%.10f\n\n", delta);
      }
    }while(delta >= deltamin);

    printf("\n\nBEST_BOUND\n");
    computeLambda(a3min,a4min,1);

    system("PAUSE");
    return 0;

}
```