Cole's Parametric Search Technique Made Practical

Michael T. Goodrich
Dept. of Computer Science
University of California, Irvine

Paweł Pszona
Dept. of Computer Science
University of California, Irvine

Abstract

Parametric search has been widely used in geometric algorithms. Cole's improvement provides a way of saving a logarithmic factor in the running time over what is achievable using the standard method. Unfortunately, this improvement comes at the expense of making an already complicated algorithm even more complex; hence, this technique has been mostly of theoretical interest. In this paper, we provide an algorithm engineering framework that allows for the same asymptotic complexity to be achieved probabilistically in a way that is both simple and practical (i.e., suitable for actual implementation). The main idea of our approach is to show that a variant of quicksort, known as *boxsort*, can be used to drive comparisons, instead of using a sorting network, like the complicated AKS network, or an EREW parallel sorting algorithm, like the fairly intricate parallel mergesort algorithm. This results in a randomized optimization algorithm with a running time matching that of using Cole's method, with high probability, while also being practical. We show how this results in practical implementations of some geometric algorithms utilizing parametric searching and provide experimental results that prove practicality of the method.

1 Introduction

Parametric search [23] has proven to be a useful technique in design of efficient algorithms for many geometric and combinatorial optimization problems (e.g., see [2, 3, 27]). Example applications include ray shooting [1], slope selection [13], computing the Fréchet distance between two polygonal curves [6, 8], matching drawings of planar graphs [5], labeling planar maps with rectangles [21], and various other matching and approximation problems (e.g., see [15, 16, 17]).

Although it has been superseded in some applications by Chan's randomized optimization technique [9, 10], for many problems (most notably, involving Fréchet distance) asymptotically best known results still depend on parametric searching.

The technique is applied to a decision problem, B, whose solution depends on a real parameter, λ, in a monotonic way, so that B is true on some interval $(-\infty, \lambda^*)$. The goal is to determine the value of λ^*, the maximum for which B is true. To achieve this goal, the parametric search approach utilizes two algorithms. The first algorithm, $\mathcal{C}$, is a sequential *decision* algorithm for B that can determine if a given λ is less than, equal to, or greater than λ^*. The second algorithm, $\mathcal{A}$, is a *generic* parallel algorithm whose inner workings are driven by "comparisons," which are either independent of λ or depend on the signs of low-degree polynomials in λ. Because $\mathcal{A}$ works in parallel, its comparisons come in batches, so there are several independent such comparisons that occur at the same time. The idea, then, is to run $\mathcal{A}$ on the input that depends on the unknown value λ^*, which will result in actually finding that value as a kind of by-product (even though we do not know λ^*, $\mathcal{C}$ can be used to resolve comparisons that appear during the execution of $\mathcal{A}$). The next step is to simulate an execution of $\mathcal{A}$ sequentially. To resolve comparisons that occur in a single step of this simulation, we can use the algorithm $\mathcal{C}$ to perform binary search among the (ordered) roots of the polynomials in λ for these comparisons, which allows us to determine signs of all these polynomials, hence, allows us to continue the simulation. When the simulation completes, we will have determined the value of λ^*. Moreover, the running time for performing this simulation is $O(P(n)T(n) + C(n)T(n)\log P(n))$, where $C(n)$ is the (sequential) running time of $\mathcal{C}$, $T(n)$ is the (parallel) running time of $\mathcal{A}$, and $P(n)$ is the number of processors used by $\mathcal{A}$.

Cole [11] shows how to improve the asymptotic performance of the parametric search technique when sorting is the problem solved by $\mathcal{A}$. His improvement comes from an observation that performing a separate binary search for each step of the algorithm $\mathcal{A}$ will often "waste" calls to $\mathcal{C}$ to resolve a relatively small number of comparisons. Rather than resolve all the comparisons of a single step of $\mathcal{A}$, he instead assumes that $\mathcal{A}$ is implemented as the AKS sorting network [4] or an optimal EREW parallel sorting algorithm [12, 18], which allows for comparisons on multiple steps of $\mathcal{A}$ to be considered at the same time (so long as their preceding comparisons have been resolved). This improvement results in a running time for the optimization problem that is $O(P(n)T(n) + C(n)(T(n) + \log P(n)))$.

From an algorithm engineering perspective, the "classical" parametric search technique (utilizing a parallel algorithm) is admittedly difficult to implement, although some implementations do exist [28, 29, 30]. Cole's improvement is even more complex, however, and we are not familiar with any implementations of his parametric search optimization.

Even without Cole's improvement, a challenge for implementing the parametric search technique is the simula-

tion of a parallel algorithm on a sequential machine. This difficulty has motivated some researchers to abandon the use of parametric searching entirely and instead use other paradigms, such as expander graphs [20], geometric random sampling [22], and ϵ-cuttings [7] (see also [2]).

Interestingly, van Oostrum and Veltkamp [30] show that, for sorting-based parametric search applications, one can use the well-known `quicksort` algorithm to drive comparisons instead of a parallel sorting algorithm. Unfortunately, as van Oostrum and Veltkamp note in their paper, Cole's improvement cannot be applied in this case. The main difficulty is that, when viewed as a kind of parallel algorithm, comparisons to be done at one level of `quicksort` become known only after all the comparisons on the level above have been resolved. Thus, comparisons cannot be pipelined in the way required by Cole's optimization when using this approach. The result, of course, is that this sets up an unfortunate tension between theory and practice, forcing algorithm designers to choose between a practical, but asymptotically inferior, implementation or an impractical algorithm whose running time is asymptotically better by a logarithmic factor.

1.1 Our Results

We show that it is, in fact, possible to implement Cole's parametric search technique in a manner that is efficient and practical (i.e., fast and easy to implement). The main idea is to use a variant of quicksort, known as `boxsort` [25], to drive comparisons (instead of sorting networks, like the complicated AKS network or an EREW parallel sorting algorithm). We apply a potential function to comparisons in the `boxsort` algorithm, which, together with a weighted-median-finding algorithm, allows us to schedule these comparisons in a pipelined fashion and achieve, with high probability, the same asymptotic running time as Cole's method, while also being practical. Moreover, we provide experimental results that give empirical evidence supporting these claims for the "median-of-lines" problem [23] and the geometric optimization problems of matching planar drawings [5] and labeling planar maps with rectangles [21].

2 Parametric Search Explained

In this section, we provide a more in-depth description of the parametric search technique. Recall that B is a problem that we want to solve. Furthermore, we restrict ourselves to the case where the generic algorithm $\mathcal{A}$ is a sorting algorithm. We require of B the following.

1. There is a *decision algorithm*, $\mathcal{C}$, which, for any value λ, resolves a comparison $\lambda < \lambda^*$ in time $C(n)$ without actually knowing λ^* (note that $C(n)$ is a function of the size of input to B). Typically, $C(n)$ is at least $\Omega(n)$, as opposed to $O(1)$ comparison time which is usual for classical sorting algorithms.
2. There is an efficient way of generating values x_i (with each x_i being either a real value or a real-valued function of λ) from an input to problem B. Ideally, it produces $O(n)$ such values.
3. For each $x_i < x_j$ comparison, the answer is determined by the sign of a low-degree polynomial in λ at $\lambda = \lambda^*$ (polynomials for different comparisons may differ).
4. *Critical values* (values λ that, based on combinatorial properties of B, have the potential of being equal to λ^*) form a subset of the set of roots of the polynomials determining answers to every possible comparison $x_i < x_j$.

Then, as a by-product of sorting values x_i, we get (directly or indirectly) the answers to all comparisons $\lambda < \lambda^*$, where λ's are roots of all comparisons $x_i < x_j$. Therefore, we are able to find λ^*.

We can solve B in the following way: generate x_i's, sort them using algorithm $\mathcal{A}$ and recover λ^* from the answer. If $\mathcal{A}$ sorts n items in $T(n)$ comparisons and each comparison is resolved in time $O\big(C(n)\big)$ (it requires determining whether $\lambda < \lambda^*$ for a constant number of roots λ), solving B this way takes time $T(n)C(n)$.

It is important to note that if there are k comparisons $x_i < x_j$, we can avoid calling $\mathcal{C}$ on every single root of their polynomials, and still resolve them all. This is because resolving $\lambda < \lambda^*$ automatically resolves comparisons for values $\lambda' \leq \lambda$ (if the result was `YES`) or $\lambda'' > \lambda^*$ (if the result was `NO`). Therefore, we can solve k comparisons in only $O(\log k)$ calls to $\mathcal{C}$, if in every iteration we use a standard median-finding algorithm (e.g., see [14]) to find the median root λ, and then resolve it by a call to $\mathcal{C}$ (each iteration halves the number of unresolved comparisons).

The above observation lies at the heart of the original parametric search, as introduced by Megiddo [23]. Note that we can group the comparisons in such a way only if they are *independent* of each other. To assure this, one chooses $\mathcal{A}$ to be a *parallel* sorting algorithm, running in time $T(n)$ on $P(n)$ processors. At every step of $\mathcal{A}$, there are $O(P(n))$ independent comparisons, and they can be resolved in time $O\big(P(n) + \log(P(n)) \cdot C(n)\big)$ according to the previous observation. Resolving comparisons at all $T(n)$ steps of $\mathcal{A}$ takes time $O\big(T(n) \cdot P(n) + T(n) \cdot \log(P(n)) \cdot C(n)\big)$. Simulating $\mathcal{A}$ on a sequential machine takes time $O\big(T(n)P(n)\big)$. Therefore, parametric search, as originally introduced, helps solve B in time $O\big(T(n) \cdot P(n) + T(n) \cdot \log(P(n)) \cdot C(n)\big)$.

2.1 Cole's Improvement

Cole [11] was able to improve on Megiddo's result by using a sorting network or an EREW parallel sorting algorithm as $\mathcal{A}$, and changing the order of comparison resolution by assigning weights to comparisons and resolving the *median weighted comparison* at each step.

In the case of a sorting network, a straightforward notion of *active* comparisons and *active* wires was introduced. Ini-

tially, all input wires (and no others) are *active*. A comparison is said to be *active* if it is not resolved and both its input wires are *active*. When active comparison gets resolved, its output wires now become *active*, possibly activating subsequent comparisons. Informally, *active* comparisons have not been resolved yet, but both of their inputs are already determined.

Weight is assigned to every comparison, being equal to 4^{-j} for a comparison at depth j. The *active weight* is defined as the weight of all *active* comparisons. The weighted median comparison can be found in $O(n)$ time [26], and resolving it automatically resolves a weighted half of the comparisons.

It is shown that for a sorting network of width $P(n)$ and depth $T(n)$, or an EREW sorting algorithm with $P(n)$ processors and time $T(n)$, the method of resolving weighted median comparison requires only $O(T(n) + \log(P(n)))$ direct calls to $\mathcal{C}$. Including simulation overhead, we solve B in time $O(P(n) \cdot T(n) + \big(T(n) + \log(P(n))\big) \cdot C(n))$.

This is completely impractical, however, as the bounds for the AKS network have huge constant factors. In a subsequent work [12], Cole shows that one can substitute an EREW parallel sorting algorithm for the AKS network, which makes using his optimization more implementable, but arguably still not practical, since the existing optimal EREW parallel sorting algorithms [12, 18] are still fairly intricate.

2.2 Applying `quicksort` to Parametric Search

Van Oostrum and Veltkamp [30] have shown that the `quicksort` algorithm [19] can be used as $\mathcal{A}$. Recall that in the randomized version of this algorithm we sort a set of elements by picking one of them (called the *pivot*) at random, and recursively sorting elements smaller than the pivot and greater than the pivot. A key observation here is that all the comparisons with the pivot(s) at a given level of recursion are independent of each other. It leads to a practical algorithm, running in $O(n \log n + \log^2 n \cdot C(n))$ expected-time, for solving B (it becomes $O(n \log n + \log n \cdot C(n))$ under additional assumption about distribution of the roots of polynomials). Comparisons are resolved by resolving the median comparison among unresolved comparisons at the current level. As `quicksort` is expected to have $O(\log n)$ levels of recursion, and $O(n)$ comparisons at each level can be resolved in time $O(n + \log n \cdot C(n))$, time bound follows.

Cole's improvement cannot be applied in this case, because all comparisons at one level have to be resolved before we even know what comparisons have to be done at the next level (that is, we don't know the splits around pivots until the very last comparison is resolved).

3 Our Practical Version of Cole's Technique

In this section, we describe our algorithm engineering framework for making Cole's parametric search technique practical. Our approach results in a randomized parametric search algorithm with a running time of $O(n \log n + \log n \cdot C(n))$, with high probability, which makes no assumptions about the input. Our framework involves resolving median-weight comparison, according to a potential function based on Cole-style weights assigned to comparisons of a fairly obscure sorting algorithm, which we review next.

3.1 The `boxsort` Algorithm

We use the `boxsort` algorithm due to Reischuk [25] (see also [24]) as $\mathcal{A}$. This algorithm is based on an extension of the main idea behind randomized `quicksort`, namely splitting elements around pivots and recursing into subproblems. While `quicksort` randomly selects a single pivot and recurses into two subproblems, `boxsort` randomly selects $\sqrt{n}$ pivots and recurses into $\sqrt{n}+1$ subproblems in a single stage. We think of it as a parallel algorithm, in the sense that the recursive calls on the same level are independent of each other. The pseudocode is shown in Algorithm 1.

```
    // N – original number of items
proc boxsort (A[i ... j])
 1: n ← (j − i + 1)
 2: if n < log N then        // base case
 3:    sort A[i ... j]
 4: else
 5:    randomly mark √n items
 6:    sort the marked items
 7:    use the marked items to split A[i ... j] into subprob-
       lems A_1, A_2, ..., A_{√n+1}
 8:    for all i ← 1 ... √n + 1 do
 9:       boxsort (A_i)
10:    end for
11: end if
```

Algorithm 1: `boxsort`

Few details need further explanation. Sorting in lines 3 and 6 is done in a brute-force manner, by comparing all pairs of items, in time $O(n^2)$ in line 3, and $O(n)$ in line 6 (note that since all these comparisons are independent, they can all be realized in a single parallel step).

Once the *marked* items are sorted in line 6, splitting in line 7 is simply done by $n - \sqrt{n}$ independent binary searches through the *marked* items (to determine, for each unmarked element, the subproblem where it lands). It takes $O(n \log \sqrt{n})$ time (when realized in a sequential way). Equivalently, we think of the sorted set of *marked* items as forming a perfectly balanced binary search tree. Locating a destination subproblem for an item is then done by *routing* the item through this tree. The tree has $\log \sqrt{n}$ levels, and all

routing comparisons are independent between different unmarked items. Therefore, *routing* can be realized in $\log \sqrt{n}$ parallel steps.

3.2 Weighting Scheme

Motivated by Cole's approach, we assign weight to every *active* comparison, and resolve the weighted median comparison in a single step. For simplicity, we identify each comparison $x_i < x_j$ with a single comparison against the optimum value, i.e., $\lambda_{ij} < \lambda^*$ for real λ_{ij} (in essence, we assume that comparison polynomials have degree 1). It is straightforward to extend the scheme for the case of higher degrees of comparison polynomials.

It makes sense here to think of `boxsort` in a network-like fashion, in order to understand how the weights are assigned to comparisons. Here, nodes represent comparisons, and directed edges represent dependence on previous comparisons. Furthermore, we imagine the network with edges directed downward, and refer to edge sources as *parents*, and destinations as *children*. Comparison becomes *active* as soon as all its dependencies become resolved (and stops when it gets resolved).

Our "network" also contains nodes for *virtual comparisons*. These are not real comparisons, and don't appear during actual execution of the algorithm. Their sole purpose is to make it easy to assign weights to *real* comparisons once they become *active* (we will later see that, in fact, they are not necessary even for that; but they make it easy to understand how the weights are computed). When a *virtual* comparison becomes *active*, it is automatically resolved (reflecting the fact that there is no *real* work assigned to a virtual comparison).

Contrary to Cole's weighting scheme for sorting networks, our scheme does not rely only on comparison's depth when assigning weights. In fact, different comparisons at the same level of the network may have different weights. Weights are assigned to comparisons (virtual or not) according to the following *weight rule*:

> When comparison C of weight w gets resolved and causes m comparisons $C_1, \ldots, C_m$ to become *active*, each of these comparisons gets weight $w/2m$.

Informally, resolved comparison distributes half of its weight among its newly activated children. Each comparison gets its weight only once, from its last resolved parent (the scheme guarantees that all parents of a comparison have equal weight).

3.3 The Algorithm

Simulating a single recursive call of `boxsort` (including the *virtual* parts) consists of the following steps.

1. Randomly *mark* $\sqrt{n}$ items.
2. Create $\sqrt{n}\cdot(\sqrt{n}-1)/2 = O(n)$ comparisons for sorting *marked* items.
3. Construct a complete binary tree of virtual comparisons (comparisons from Step 2 are leaves).
4. Create *routing* trees from section 3.1 for routing unmarked elements; make the root of each such tree depend on the root of the tree from Step 3.
5. Route items through the tree of *marked* items;
6. Construct a binary tree of virtual comparisons (leaves are last comparisons from *routing* trees).
7. Split items into boxes
8. Assign weights for comparisons in the next level of recursion (after the items are split into boxes) by making them children of the root from Step 6.
9. Recurse into subproblems (simultaneously).

Colors represent *virtual* parts of the algorithm and correspond to the pictorial explanation of the algorithm from the appendix (Figure 1). Blue steps (3, 6) deal with trees of virtual comparisons, while red steps (4, 8) represent relationships that make *real* comparisons depend on *virtual* ones. The idea behind blue steps is to ensure synchronization (that is, guarantee that all *real* comparisons on the levels above have been resolved), and red steps are there to ensure proper assignment of weights. Figure 1 also shows heights of the trees used and weights assigned to comparisons on levels of the "network". For simplicity, it presents heights/weights as if there were exactly n (instead of $\sqrt{n}\cdot(\sqrt{n}-1)/2$) comparisons between *marked* items, and exactly n (instead of $n-\sqrt{n}$) unmarked items to be routed. The following discussion is also based on this assumption.

Steps 1 and 7 do not involve any comparisons, and they do not affect weights. Comparisons from Step 2 start with weight w. The tree from Step 3 has height $\log n$, so its root, according to the *weight rule* gets weight $w/(2^{\log n}) = w/n$. Dependencies introduced in Step 4 between that root and roots of the *routing trees* cause their weight to be $w/2n^2$ (weight w/n divided among n comparisons). *Routing trees* have height $\log \sqrt{n}$, so the comparisons at their bottom have weight $w/2n^{2.5}$ ($w/2n^2$ divided by $2^{\log \sqrt{n}}$, because, as the routing progresses, the *routing trees* get whittled down to paths, and resolving a routing comparison *activates* at most one new routing comparison. Step 6 is essentially the same as Step 3, so the root of the second *virtual* tree gets weight $w/2n^{3.5}$. All initial comparisons in the subsequent recursive calls (sorting of new *marked* items and/or sorting in the base case) depend on this root (Step 8), and they are given weight $w/4n^{4.5}$ (much like in Step 4). The height of the dependence network is $O(\log n)$, and at any given moment the number of currently *active* comparisons does not exceed n.

From now on, comparisons are independent across different subproblems. For subsequent subproblems, n from the above discussion gets substituted by $\hat{n}$, the size of the subproblem. Since subproblem sizes may differ, comparisons on the same level of the network (general level, for the entire algorithm) are no longer guaranteed to have same weights (weights of comparisons belonging to the same subproblem

are however equal).

The above discussion shows that, as advertised, we don't really need *virtual* comparisons in order to assign weights to *real* comparisons, as these depend only on n, the size of the subproblem. Therefore, the actual algorithm only consists of steps 1, 2, 5, 7, and 9 and is the following.

1. Randomly *mark* $\sqrt{n}$ items
2. Sort *marked* items by comparing every pair in $O(n)$ comparisons, each of weight w.
3. When the last comparison finishes, *activate* comparisons for routing through the tree of *marked* items, each of weight $w/2n^2$.
4. Route items through the trees, following the *weight rule* when a comparison gets resolved.
5. When the destination for the last item is determined, split items into boxes (no additional comparisons resolved here).
6. Assign weight $w/4n^{4.5}$ to initial comparisons in new subproblems.
7. Recurse into subproblems (simultaneously).

3.4 Analysis

Assume that initially all comparisons at the highest level were given weight 1. In this analysis, we also include *virtual* comparisons. If the current *active* weight (sum of weights of all *active* comparisons) is equal to W, resolving the weighted-median comparison reduces *active* weight by at least $W/4$ (it resolves comparisons of total weight $\geq W/2$, and each resolved comparison passes only at most half of its weight to its children). Thus, the following lemma is proved identically as Lemma 1 of [11] (a turn consists of resolving median weighted comparison and assigning weights to newly activated comparisons).

Lemma 1 *At the start of the (k+1)-st turn, active weight is bounded from above by* $(3/4)^k n$*, for* $k \geq 0$*.*

We also have the following.

Lemma 2 *Each comparison at depth* j *has weight* $\geq 4^{-j}$*.*

Proof. We prove this by induction on the depth of the `boxsort` recursion. Assume that the current recursive call operates on a subproblem of size n, comparisons at the beginning of the current recursive call have depth k and weight w. By inductive assumption, $w \geq 4^{-k}$.

Consider comparisons in the current recursive call. Comparisons at depth i in the first tree of *virtual* comparisons (global depth $k+i$) have weight $w/2^i \geq 4^{-k} \cdot 2^{-i} \geq 4^{-(k+i)}$. The last of them has (local) depth $\log n$ and weight w/n. It then spreads half of its weight to n comparisons at depth $\log n + 1$ (global depth $k + \log n + 1$), setting their weight to $w/2n^2 \geq w/4n^2 = w/4^{\log n+1} \geq 4^{-(k+\log n+1)}$. The same reasoning follows for the case of the second *virtual* tree and recursive split, while routing through the tree of sorted *marked* items always decreases weight by a factor of 2 for the next level instead of 4 (making the result even stronger).

To finish the proof, note that the base case is realized in the very first call to the algorithm, since comparison at depth 0 has weight $1 = 1/4^0$. □

Lemma 2 allows us to prove the following lemma exactly as Lemma 2 of [11].

Lemma 3 *For* $k \geq 5(j + 1/2 \cdot \log n)$*, during the (k+1)-st turn there are no active comparisons at depth* j*.*

This leads to the following corollary.

Lemma 4 *If the* network *for* `boxsort` *has height* $f(n)$*,* $O(f(n)+\log n)$ *rounds of resolving the median-weight comparison suffice to resolve every comparison in the network.*

We also have the following fact about `boxsort`.

Lemma 5 *(Theorem 12.2 of [24]) There is a constant* $b > 0$ *such that* `boxsort` *terminates in* $O(\log n)$ *parallel steps with probability at least* $1 - \exp(-\log^b n)$*.*

Originally, `boxsort` requires $O(\log n)$ parallel steps to execute a single recursive call for a problem of size n. We noted that the dependence network for a single recursive call in our simulation has height $O(\log n)$ for a problem of size n as well. This means that Lemma 5 applies here and proves that, with high probability, the dependence network for the entire simulation has height $O(\log n)$.

Combining that with Lemma 4 and the observation that any level in the dependence network contains $O(n)$ comparisons, we get the following.

Theorem 6 *With high probability, the presented algorithm requires* $O(\log n)$ *calls to* $\mathcal{C}$*, yielding an* $O(n \log n + \log n \cdot C(n))$ *time parametric search solution to problem* B*.*

4 Conclusion

We have introduced a practical version of Cole's optimization of the parametric search technique. Our method results in a randomized algorithm whose running time matches that of using Cole's technique, with high probability, while being easily implementable. We have implemented it and, based on experimentation performed on some geometric problems (details in the appendix), showed that our approach is competitive with the previous practical parametric search technique of van Oostrum and Veltkamp [30], while having superior asymptotic performance guarantees.

References

[1] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM Journal on Computing*, 22(4):794–806, 1993.

[2] P. K. Agarwal and M. Sharir. Efficient algorithms for geometric optimization. *ACM Comput. Surv.*, 30(4):412–458, 1998.

[3] P. K. Agarwal, M. Sharir, and S. Toledo. Applications of parametric searching in geometric optimization. *J. Algorithms*, 17(3):292–318, 1994.

[4] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in c log n parallel steps. *Combinatorica*, 3:1–19, January 1983.

[5] H. Alt, A. Efrat, G. Rote, and C. Wenk. Matching planar maps. *Journal of Algorithms*, 49(2):262–283, 2003.

[6] H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *Int. J. Comput. Geometry Appl.*, 5:75–91, 1995.

[7] H. Brönnimann and B. Chazelle. Optimal slope selection via cuttings. *Computational Geometry: Theory and Applications*, 10(1):23–29, 1998.

[8] E. W. Chambers, E. Colin de Verdière, J. Erickson, S. Lazard, F. Lazarus, and S. Thite. Walking your dog in the woods in polynomial time. In *24th ACM Symp. on Computational Geometry*, pages 101–109, 2008.

[9] T. M. Chan. Geometric applications of a randomized optimization technique. *Discrete & Computational Geometry*, 22(4):547–567, 1999.

[10] T. M. Chan. An optimal randomized algorithm for maximum tukey depth. In J. I. Munro, editor, *SODA*, pages 430–436. SIAM, 2004.

[11] R. Cole. Slowing down sorting networks to obtain faster sorting algorithms. *J. ACM*, 34(1):200–208, 1987.

[12] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17:770–785, August 1988.

[13] R. Cole, J. S. Salowe, W. L. Steiger, and E. Szemerédi. An optimal-time algorithm for slope selection. *SIAM Journal on Computing*, 18(4):792–810, 1989.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[15] C. A. Duncan, M. T. Goodrich, and E. A. Ramos. Efficient approximation and optimization algorithms for computational metrology. In *8th ACM-SIAM Symp. on Discrete algorithms (SODA)*, pages 121–130, 1997.

[16] H. Fournier and A. Vigneron. A deterministic algorithm for fitting a step function to a weighted point-set. *CoRR (arXiv ePrint)*, abs/1109.1152, 2011.

[17] M. Goodrich. Efficient piecewise-linear function approximation using the uniform metric. *Discrete & Computational Geometry*, 14:445–462, 1995.

[18] M. T. Goodrich and S. R. Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *J. ACM*, 43:331–361, March 1996.

[19] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4:321–, July 1961.

[20] M. J. Katz and M. Sharir. Optimal slope selection via expanders. *Information Processing Letters*, 47(3):115–122, 1993.

[21] A. Koike, S.-I. Nakano, T. Nishizeki, T. Tokuyama, and S. Watanabe. Labeling points with rectangles of various shapes. *International Journal of Computational Geometry and Applications*, 12(6):511–528, 2002.

[22] J. Matoušek. Randomized optimal algorithm for slope selection. *Information Processing Letters*, 39(4):183–187, 1991.

[23] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30(4):852–865, 1983.

[24] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[25] R. Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM J. Comput.*, 14(2):396–409, 1985.

[26] A. Reiser. A linear selection algorithm for sets of elements with weights. *Inf. Process. Lett.*, 7(3):159–162, 1978.

[27] J. S. Salowe. Parametric search. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry, Second Edition*, pages 969–982. Chapman & Hall/CRC Press, Inc., 2004.

[28] J. Schwerdt, M. H. M. Smid, and S. Schirra. Computing the minimum diameter for moving points: An exact implementation using parametric search. In *ACM Symp. on Computational Geometry*, pages 466–468, 1997.

[29] S. Toledo. *Extremal Polygon Containment Problems and Other Issues in Parametric Searching*. MS Thesis, Dept. Comput. Sci., Tel Aviv Univ., Tel Aviv, 1991.

[30] R. van Oostrum and R. C. Veltkamp. Parametric search made practical. *Computational Geometry: Theory and Applications*, 28(2-3):75–88, 2004.

A Appendix

A.1 Pictorial Illustration of the Algorithm

Figure 1 presents the network-like interpretation of the algorithm, including *virtual* comparisons. It is meant to be helpful in understanding the description of the algorithm. A few things to keep in mind when reading it:

- Nodes represent comparisons.
- If two comparisons are linked by an edge, the lower one depends on the upper one.
- **Black** nodes represent *real* comparisons (ones that actually appear during algorithm execution).
- **Blue** nodes represent *virtual* comparisons.
- **Red** edges represent assigning weight to *real* comparisons based on *virtual* ones.
- Figure 1 shows two consecutive levels of recursion of the algorithm; first level is shown in full; only one subproblem from the second level (of size $\hat{n}$) is shown.
- For simplicity, we assume (as we did in 3.3) that there are exactly n sorting comparisons, and exactly n unmarked items are to be routed through the trees.

A.2 Experimental Results

In order to measure the performance of our method we implemented two general parametric search frameworks: our `boxsort`-based method and the `quicksort`-based algorithm of van Oostrum and Veltkamp [30]. We then implemented three known algorithms that utilize parametric search, and compared the running times of both approaches when used in these algorithms (answers were obviously the same). We start with the description of the problems and implemented algorithms.

A.2.1 Implemented Algorithms

Point labeling. The input to the *point labeling* problem is a set of points in the plane (*objects*), and sets of rectangles (*labels*), one set per point (their elements are called *candidate labels*). A *feasible* solution is one where each *object* is assigned a *label* (from its *candidate labels*), the *labels* are drawn "near" their *objects*, and they do not overlap. The goal is to find a label placement and the largest *scaling factor* $\sigma > 0$, so that the solution is still *feasible* when dimensions of *labels* are multiplied by σ. The problem is motivated by applications in Geographic Information Systems, where the goal is to label objects with largest non-overlapping labels (to improve readability).
Koike *et al.* [21] gave an algorithm for the special case when "near" is defined as "each *label* has a special *pinning point*, and *label* has to be placed so that its *pinning point* is exactly over the *object*". They show that if the *candidate labels* meet an additional requirement on the relative placement of their *pinning points*, it can be decided if *candidate labels* scaled by a given σ have a *feasible* placement, in time $O(n \log n)$, by a variant of the plane sweep method, where n is the number of *objects*. They use this algorithm as the decision algorithm, $\mathcal{C}$, for parametric search. Algorithm $\mathcal{A}$ from the parametric search setting is defined as sorting the (symbolic) coordinates of labels for σ^* – the optimal value of the *scaling factor*. Therefore, their parametric search algorithm for finding σ^* works in $O(n \log^2 n)$ time.
An example is shown in Figure 2.

Matching planar drawings of graphs. This problem was defined by Alt *et al.* [5], and is a generalization of the problem of computing the Fréchet distance between two polygonal curves to computing a Fréchet-like distance between two graphs, G and H, embedded on a plane.
A common explanation of the Fréchet distance between two curves, g and h, is the following: "A man is walking his dog. What is the smallest length of the leash that allows the dog to traverse entire h, while simultaneously the man traverses entire g?" In the man-dog setting, the generalized problem for graphs G and H is stated the following way: "What is the smallest length of the leash that allows the dog to traverse the entire graph H, while simultaneously the man traverses *some* part of G?" This intuition translates into a parametric transformation that can, for example, be used to produce a minimum-deformation morphing of one straight-line graph drawing into another.
Alt *et al.* gave an $O(pq \log(pq))$ algorithm for deciding if a given λ, leash length, is sufficient, where p, q are respective numbers of edges in G and H. They construct a graph that relates possible movements of the man along edges of G and the movements of the dog along edges of H with a leash of length λ. Then they perform a search in this graph to find a path that describes contiguous movement of man and dog that covers the entire H. It is used as the algorithm $\mathcal{C}$ from the generic parametric search setting. $\mathcal{A}$ is again defined as sorting items that describe the features of the aforementioned graph for optimal length of the leash, λ^* (see [5] for details). It yields an $O(pq \log^2(pq))$-time parametric search algorithm for finding λ^*.
An example is shown in Figure 3.

Median of lines. This "toy problem" was used by Megiddo to introduce and explain the parametric search technique in the original paper on the subject [23]. The input consists of n lines on a plane and the goal is to find a line whose intersection with the x-axis has the same number of lines directly above and below it. Megiddo gave a simple (although non-optimal) algorithm that utilizes parametric search and works in time $O(n \log n)$.
An example is shown in Figure 4.

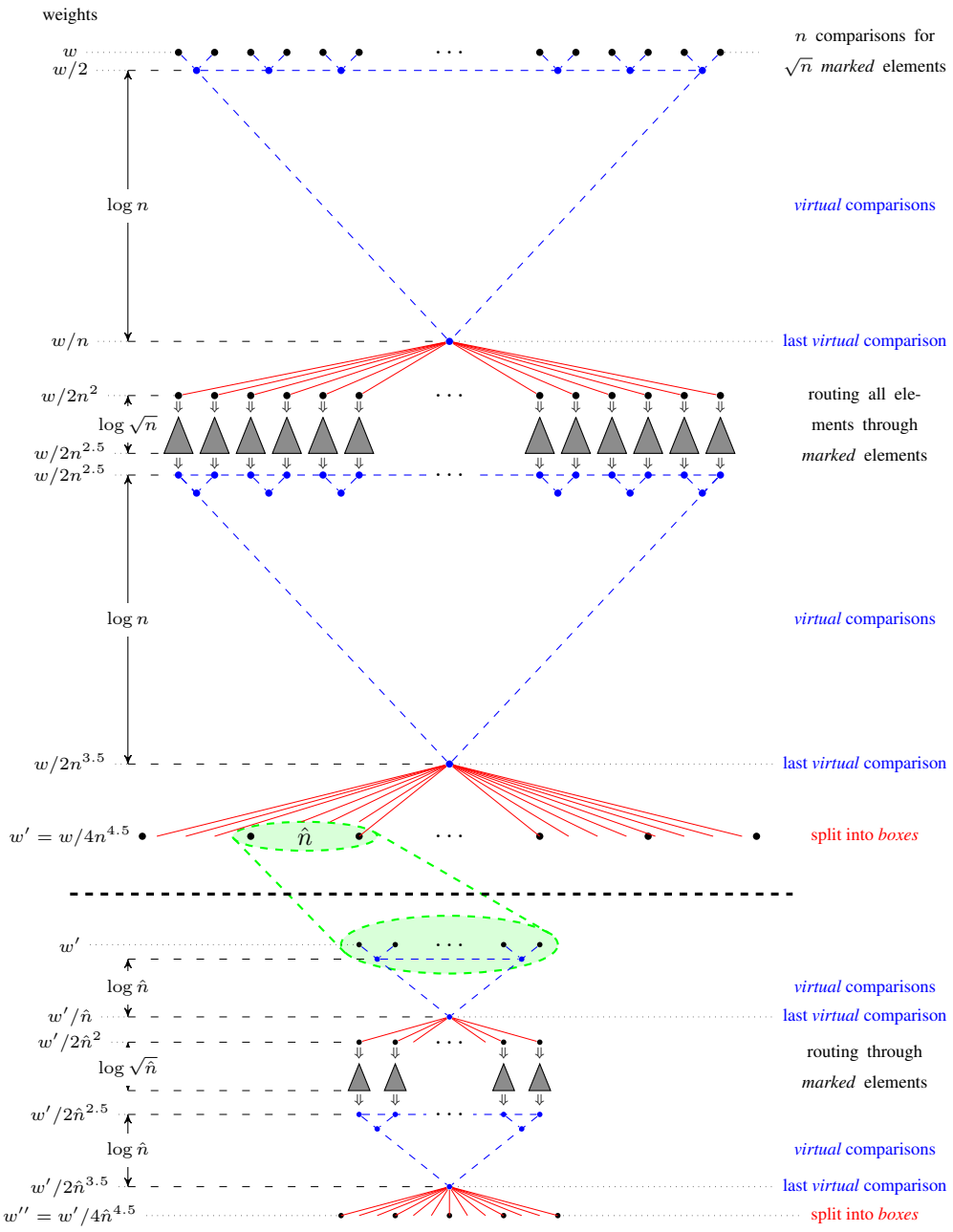


Figure 1: Algorithm and assigned weights – first 2 levels of recursion

A.2.2 Testing

Algorithms for the above problems were implemented in C++, and tested on random inputs on two PC's running on Linux:

- median of lines: 800 MHz CPU, 4 GB memory
- point labeling and matching planar drawings of graphs: 2.50 GHz CPU, 4 GB memory.

For the point labeling problem, inputs were n *objects* with integer coordinates randomly chosen from 0-10^6, with 6 *candidate labels* each, pinned at their lower left corners. For the graph matching problem, inputs were two graphs with n vertices and $O(n)$ edges each. Vertices were placed at points with rational coordinates, with numerator and denominator drawn independently from 1-10^4 (the complexity of the algorithm being $O(n^2 \log^2 n)$ in this case). For the median of lines problem, inputs were n lines defined by the $y = ax + b$ equation, with randomly chosen rational a and b values. We ran the tests on numerous values of n, doing 100 tests for each value of n. The results are presented in Tables 1, 2 and 3.

input size (n)	boxsort-based					quicksort-based				
	time [ms]			avg	avg	time [ms]			avg	avg
	avg	min	max	comp	resol	avg	min	max	comp	resol
10	0.8	0	2	11.4	7.2	0.6	0	2	11.7	6.7
100	40.5	33	48	43.6	14.3	37.9	30	56	63.1	15.4
500	323.1	282	365	69.2	20.0	322.8	271	410	129.2	20.5
1000	736.4	644	809	79.3	22.2	753.2	610	919	163.9	23.5
2000	1674.1	1474	1852	89.4	24.6	1720.6	1475	2047	206.9	26.4
5000	4788.9	4353	5386	102.7	27.5	4986.3	4322	5838	268.0	28.4
8000	8194.6	7366	9218	111.1	30.2	8648.8	7516	10252	303.7	31.1
10000	10480.5	9189	11843	113.3	30.4	10986.5	9773	13210	323.8	29.7
12000	13081.8	11995	14688	116.7	31.8	13855.2	12272	16810	334.1	32.4
14000	15532.0	14127	17382	118.7	32.4	16233.5	14695	18813	346.4	32.7
16000	17926.0	15925	19975	120.4	31.8	18914.1	16779	21375	360.8	32.5
18000	20618.1	18175	23183	121.3	31.8	21764.0	19081	24505	369.1	33.1
20000	23413.6	21516	26015	124.2	32.6	24406.6	20939	28470	377.4	34.2

Table 1: Test results for the median-of-lines problem; avg comp – average number of median comparison resolutions; avg resol – average number of median comparisons actually resolved using $\mathcal{C}$

input size (n)	boxsort-based					quicksort-based				
	time [s]			avg	avg	time [s]			avg	avg
	avg	min	max	comp	resol	avg	min	max	comp	resol
10	0.003	0.001	0.004	33.3	11.0	0.003	0.001	0.005	49.0	10.4
100	0.027	0.012	0.037	69.9	17.1	0.023	0.010	0.037	141.5	17.2
500	0.283	0.133	0.428	97.4	23.1	0.258	0.116	0.466	238.8	21.7
1000	1.167	0.515	2.352	106.4	22.8	1.104	0.511	1.906	281.4	23.0
1500	2.355	0.712	4.466	113.3	23.5	2.122	0.813	4.914	325.8	19.1
2000	4.628	2.002	9.222	116.5	20.2	4.335	1.475	8.427	344.0	22.8
3000	11.619	3.151	22.068	121.6	21.2	9.466	2.609	17.662	365.4	18.8
4000	18.461	7.910	36.385	124.9	19.6	17.301	8.274	30.196	409.6	21.7
5000	30.618	9.844	56.330	124.7	18.2	25.990	9.430	49.221	425.9	20.8
6000	42.180	14.767	73.373	128.0	21.6	35.607	18.670	63.048	433.6	17.5
7000	58.630	15.824	137.950	132.3	19.0	48.270	12.282	113.416	447.9	19.9
8000	77.023	23.396	128.263	133.6	18.6	66.104	25.622	123.445	459.2	21.6
9000	95.203	44.364	174.155	132.6	18.7	73.835	28.754	131.519	464.8	17.7
10000	105.006	37.494	215.619	134.6	19.8	96.872	38.702	187.137	474.9	20.4
11000	136.610	30.205	254.993	136.3	18.1	122.691	47.178	241.167	488.9	17.5
12000	168.988	63.704	335.881	138.3	19.8	139.351	68.135	239.619	496.0	17.8
13000	192.149	59.290	399.326	135.5	21.2	162.257	49.541	376.480	505.2	19.3
14000	209.349	70.579	461.083	138.8	18.4	178.618	63.412	372.113	524.4	17.8
15000	243.451	73.700	405.206	139.7	16.5	208.025	66.146	409.933	518.0	18.2

Table 2: Test results for the point labeling problem; avg comp – average number of median comparison resolutions; avg resol – average number of median comparisons actually resolved using $\mathcal{C}$

Recall that comparisons are of the form $\lambda < \lambda^*$. If we have previously determined that $\lambda_1 < \lambda^*$ ($\lambda_1 > \lambda^*$), and comparison $\lambda_2 < \lambda^*$ is the new median comparison we want resolved, there is no need to invoke $\mathcal{C}$ when $\lambda_2 < \lambda_1$ ($\lambda_2 > \lambda_1$), as the answer is obvious. Tables 1, 2 and 3, show both the number of times the algorithms needed to call $\mathcal{C}$, and the number of total median comparisons the algorithms wanted resolved (the difference being the number of immediately resolved median comparisons).

input size (n)	boxsort-based					quicksort-based				
	time [s]			avg	avg	time [s]			avg	avg
	avg	min	max	comp	resol	avg	min	max	comp	resol
4	0.038	0.014	0.070	77.1	18.8	0.032	0.012	0.057	125.8	16.7
8	0.465	0.169	0.838	122.5	26.4	0.393	0.152	0.686	273.1	22.8
12	1.853	0.875	3.049	147.8	29.7	1.692	0.771	2.777	367.5	27.1
16	4.861	1.944	7.269	163.3	33.6	4.473	1.702	6.866	433.8	29.5
20	9.552	5.193	13.527	173.9	35.3	9.064	4.469	11.916	438.7	32.0
24	16.145	8.633	21.313	181.9	37.4	15.415	7.717	20.654	541.3	33.5
28	23.892	10.435	32.191	194.0	38.5	23.817	10.606	31.826	579.1	36.2
32	35.894	22.690	48.253	205.7	42.0	35.064	15.580	46.285	620.1	38.1
36	46.377	29.302	56.620	211.6	38.7	47.254	27.589	63.524	658.3	37.2
40	62.825	39.393	78.046	217.1	42.3	63.635	37.847	77.617	682.3	39.3
44	79.575	54.912	100.601	214.9	41.9	80.520	54.202	101.100	700.9	40.3
48	99.633	67.896	123.148	220.8	43.1	100.102	62.814	124.495	726.1	39.4
52	122.606	77.710	156.918	215.4	44.2	122.633	78.534	150.289	738.7	39.1
56	148.018	98.792	191.290	225.7	43.9	149.623	94.775	186.019	766.2	40.1

Table 3: Test results for the graph matching problem; avg comp – average number of median comparison resolutions; avg resol – average number of median comparisons actually resolved using $\mathcal{C}$

A.2.3 Test Summary

Test results reveal some interesting properties of the underlying problems.

For the point labeling problem, both algorithms perform about 20 calls to $\mathcal{C}$, regardless of the input size. In this case the `quicksort`-based algorithm works about 15% faster.

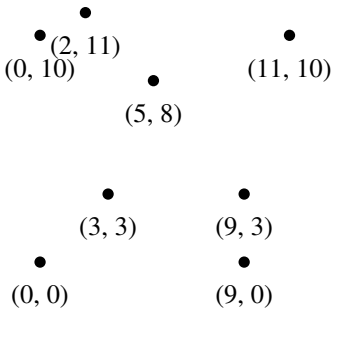
For the median-of-lines and graph matching problems, the required number of calls to $\mathcal{C}$ grows steadily as input size increases. In these cases, both algorithms run in virtually the same time, with the `boxsort`-based algorithm seemingly gaining advantage as input size grows.

We note that the `quicksort`-based algorithm might be favored by our choice of input data, as it was shown [30] that for sufficiently random input (defined as having close to uniform distribution of the roots of comparison polynomials), the algorithm requires only $O(\log n)$ calls to $\mathcal{C}$.

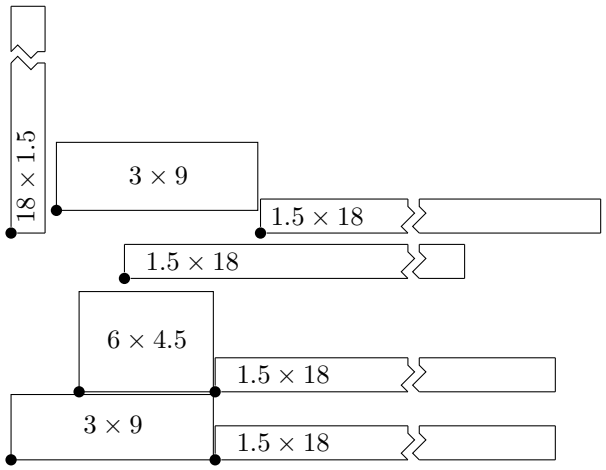
Our algorithm, on the other hand, always requires only $O(\log n)$ calls to $\mathcal{C}$ with high probability, *regardless of the input*. Based on the performed tests, we can say that our solution is practically competitive with the `quicksort`-based method, while having superior provable bounds on the running time.

A.3 Example Instances of Implemented Problems

In Figures 2, 3 and 4, we show sample inputs and results for the test problems.



a)



b)

Figure 2: Illustration of the performance of the point labeling algorithm.
a) *objects* for the point labeling problem; *candidate labels* for each point are rectangles of proportions (width×height): 1×12, 2×6, 3×4, 4×3, 6×2 and 12×1.
b) *feasible* placement of labels with scaling factor $\sigma = 1.5$.

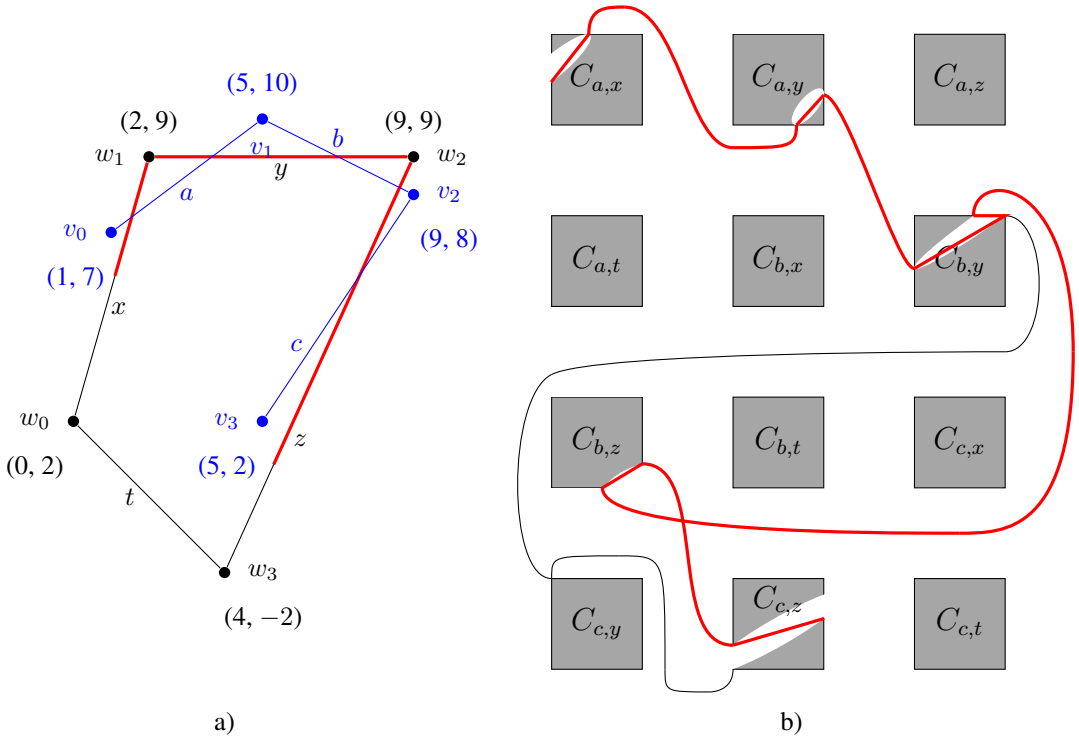


Figure 3: Illustration of the graph matching algorithm.
a) Input to the problem. Graph H is drawn in blue, while graph G in drawn in black. A "man's path" that allows the "dog" to visit all of H with a leash of (optimal) length 1 is marked in red.
b) Linked *free space* diagrams for the decision algorithm with a leash of length 1 form a graph (one $C_{g,h}$ diagram for each pair of edges (g, h), $g \in G$, $h \in H$; white area represents *free space*; edges connect respective facets of diagrams reachable via *free space*); for details, refer to [5]. Path corresponding to the "man's path" from a) is marked in red.

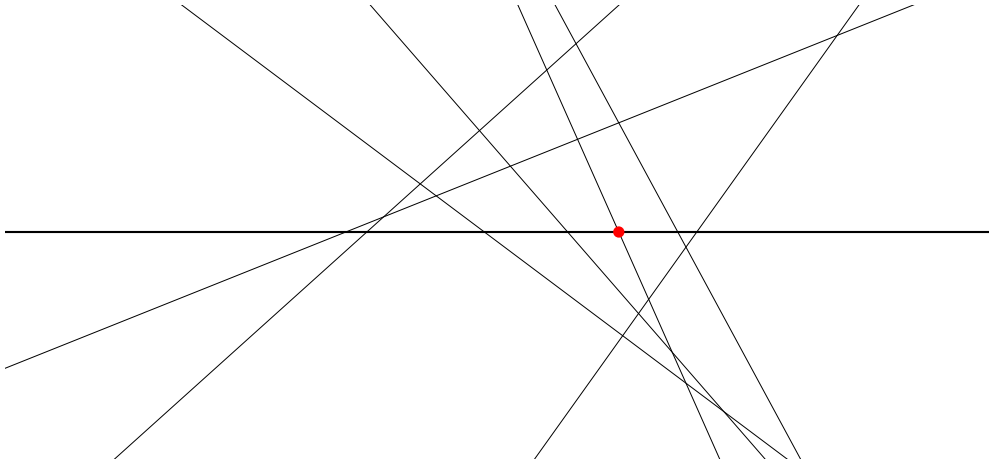


Figure 4: Illustration of the median-of-lines problem with 7 lines. Median point is marked in red (there are 3 lines above, and 3 lines below it).