The Algorithms

In this section we will describe the algorithms we use for each of the placement models. All our algorithms use a quad tree for quick finding of points in an interval, this will be described first. Furthermore, the algorithms for the 2pos model and the 4pos model are very similar, these will be discussed together. After that, we will describe the algorithm for the 1slider model.

Quadtree

Since advanced label-placing algorithms have a tendency of requesting information about what labels exist in a certain area, a quadtree can help speed up those algorithms. A quadtree has four children per node and each node of the tree stores information about a certain square-shaped area it patrols. All labels stored in a node have their origin point in the area of that node. Furthermore, the four child nodes are the subdivision of its parent node. This means that nodes in a certain area can be easily found by traversing the tree.
A quadtree implementation is being worked on, but is not yet finished. None of the previously described algorithms use the quadtree functionality yet.

Algorithm 1 Your Algorithm Here

```
1: procedure 1SLIDER(points[], labelwidth, labelheight)
2:     r ← a mod b
3:     while r ≠ 0 do                          ▷ Your comment here
4:         a ← b
5:         b ← r
6:         r ← a mod b
7:     end while
8:     return b                                ▷ Your comment here
9: end procedure
```

2-position model and 4-position model

For the 2-position model and the 4-position model we use a very simple approach. We will first our algorithm. While doing this we will also derive the theoretical running times. Then we will prove that it does not place overlapping labels. After that we will prove this approach applies the rules L1, L2 and L3 (from Automated Label Placement in Theory and Practice, by Alexander Wolff) correctly.

Description of the algorithm

Our algorithm is simple but also very efficient. It has two phases:

1. **Phase 1: Mapping the collisions**
   It checks for every candidate (a candidate is a possible placement of a label for a point) with which candidates it intersects. For each candidate, we store every candidate it collides with in an arraylist, the so-called CollisionList of a candidate. We find these collisions using the quad tree. When we have these collisions, we store all candidates in a MinHeap. The nodes are the candidates and the keys are their number of collisions (plus 0.25 times the number of candidates its point has). Using this data structure is very useful later on, when we want to extract the minimum very often and change keys of nodes even more frequently.

2. **Phase 2: Placing candidates**
   It may be shocking how easy our actual approach is: we find the candidate with the least amount of collisions (with the MinHeap) and we place it. We place a candidate using the Place function. This function takes a candidate c as argument. It will do the following: It will eliminate all the candidates of the point of which c is a candidate and it will also eliminate all the candidates that c collides with (every candidate in c's CollisionList). This eliminating will be done using an Eliminate function. This function takes a candidate c as argument and it will remove c from all the CollisionLists for all candidates c^* in the CollisionList of c. It will also remove c from the MinHeap. In this way, later on will not have to look after candidates that are already placed or are eliminated.

Below you can see the pseudo-code for the algorithms. First we will show the main algorithm and then we will show the smaller functions, such as the Place and Eliminate function.

Algorithm 2 Main Algorithm

1: **procedure** K-Pos-Algorithm ▷ Can be used for 2pos, 4pos or any k-pos model
2: Initialize the solution S to the empty set
3: Insert all points in the quad tree q
4: Give points their candidates
5: **for** $c \in$Candidates **do** ▷ For each point we will find its collisions
6: $c.CollisionList \leftarrow$ GiveCollisions(c)
7: **end for**
8: Create a MinHeap h containing all the candidates
9: **while** H is not empty **do**
10: $S \leftarrow S \cup \{h.\text{minimum}\}$ ▷ Add the candidate with the least collisions to the solution
11: Place(h.Minimum) ▷ We place this minimum
12: **end while**
13: **return** S ▷ S is our solution, we will return it
14: **end procedure**

Running time

The first line takes $\Theta(1)$ of course, the second line will take $\Theta(n \log n)$ since it inserts (insertion takes $\Theta(\log n)$) n points. For the third line it has to give n points k candidates. Giving a point a candidate takes $\Theta(1)$. In the case of the 4-position and the 2-position model, k equals 4 and 2 respectively. So this line will take $\Theta(n)$. Now there is the for-loop. This loops over all $\Theta(n)$ candidates and gives it its collisions. The latter takes $\Theta(n)$ as described below. Now we create a MinHeap, creating a Heap takes $\Theta(n)$. Now there is a while-loop. In every cycle it gets a maximum and places a label. When it places a label, it eliminates at least all the candidates of one point. So this loop iterates at most n times. In this loop, we get the minimum ($\Theta(\log n)$ for a MinHeap) and place it ($\Theta(n^2)$ as described below). The return statement takes $\Theta(1)$. So the whole 2-position and 4-position algorithm takes $\Theta(1) + \Theta(n \log n) + \Theta(n) + \Theta(n) * \Theta(n) + \Theta(n) + \Theta(n) * (\Theta(1) + \Theta(n^2)) = \Theta(n^3)$.

Algorithm 3 The algorithm that returns the colliding candidates of a candidate c

```
1: procedure GiveCollisions(Candidate c)
2:     Initialize the CollisionsList C to the empty set
3:     PossibleCollisions← q.LabelInArea(c) ▷ This will return all points that can have
   an intersecting candidate
4:     for p ∈PossibleCollisions do                    ▷ We go through all these points
5:         for each candidate c_p of p do
6:             if c_p intersects c then
7:                 add c_p to C
8:             end if
9:         end for
10:    end for
11:    return C                ▷ C now contains all the candidates that collide with c
12: end procedure
```

Running time

The first statement takes $\Theta(1)$. The second statement uses the LabelsInArea function, this has running time in the order of $\log n$ plus the number of points in the area. The number of points is $\Theta(n)$ (with large enough labels and dense enough clusters, every point can have a candidate intersecting c). So this results in $\Theta(n)$. Then there is the loop going through all these $\Theta(n)$ points. Inside this loop it checks for each candidate of the point (this number of candidates is $\Theta(1)$) whether it intersects with c, this takes $\Theta(1)$. Adding it also takes $\Theta(1)$. So concluding: The running time of this procedure is $\Theta(1) + \Theta(n) + \Theta(n) * \Theta(1) * (\Theta(1) + \Theta(1)) = \Theta(n)$.

Algorithm 4 The algorithm that places a candidate c

```
1: procedure Place(Candidate c)
2:     for Every candidate c_p of the point of c do
3:         Eliminate(c_p)          ▷ Eleminate all candidates of the point that we place
4:     end for
5:     for c_c ∈ c.CollisionList do
6:         Eliminate(c_c) ▷ Also eliminate all candidates that collide with the candidate
   we are placing
7:     end for
8: end procedure
```

Running time

There are two loops in this procedure. The first loops over all the candidates of the point of c. So 4 for the 4-position model and 2 for the 2-position model. $\Theta(1)$ either way. In this loop it performs the Eliminate function, this will take $\Theta(n)$ as you can see below. The second loop loops over all the candidates that collide with c, these are at most $k * n$ in the worst case. So $4n$ for the 4-position model and $2n$ for the

2-position model. $\Theta(n)$ in both cases. Inside this loop, we perform Eliminate again, this again takes $\Theta(n)$ every time. So the total running time of this procedure will be $\Theta(1) * \Theta(n) + \Theta(n) * \Theta(n) = \Theta(n^2)$

Algorithm 5 The algorithm that eliminates a candidate c

```
1: procedure ELIMINATE(Candidate c)
2:     for c_c ∈ c.CollisionList do
3:         Remove c from c_c.CollisionList ▷ let c_c know it does not have to look after c anymore
4:     end for
5:     remove c from the MinHeap
6: end procedure
```

Running time

As you can see there is one loop in this procedure. This loop loops over all collisions of c, the number of collisions is $\Theta(n)$. Removing the candidate from the CollisionList takes $\Theta(1)$. Furthermore, removing a node from a MinHeap takes $\Theta(\log n)$ of course. Concluding: The whole procedure takes $\Theta(n) * \Theta(1) + \Theta(\log n) = \Theta(n)$

Proof of no overlapping labels

We will prove the correctness (no intersections) of our algorithms using a loop invariant over the while-loop in the main algorithm:

Loop invariant: S does not contain overlapping candidates. Futhermore, there is no candidate $c_s \in S$ that collides with a candidate c_h in the heap.

Initialization: S is empty initially, so there are no two candidates that overlap each other and there is also no candidate in S that overlaps any of the candidates in the heap.

Maintenance: In every iteration of the while-loop, exactly one candidate c_m is added to S. Because of the loop invariant, it has no intersections with any of the candidates that were already in S. This proves the maintenance of the first part of the loop invariant. When placing the candidate, we eliminate itself and all its collisions. Calling the Eliminate function on a candidate c will remove c from the MinHeap. So all candidates in the Heap that collide with c_m will be removed from the Heap. This together with the second part of the loop invariant (no candidate $c_s \in S$ collides with a candidate c_h in the heap) proves the maintaince of the second part of the loop invariant.

Termination: When the loop is finished, our loop invariant proves that there are no intersecting candidates in S.

Proof of correctly applying the rules

Our algorithm may look too simple. But in its simplicity lies its elegance. We wanted an algorithm that could place labels efficiently in unclustered areas (using the rules) and also could give a reasonably good solution for clustered areas. This both is done by this very simple approach. Below we will prove that this applies the rules correctly.

Rule L1
L1 applies if some p has a candidate p_i without any overlaps, declare p_i to be part of the solution, and eliminate all other candidates of p.

We will prove that our algorithm places a label for p when L1 occurs. For the sake of contradiction, we will assume no label for p is placed after termination. After termination, the heap is empty. So p_i must have been removed from the heap. This only occurs when a candidate of p has been placed (which we assumed is not the case) or when a label has been placed that overlaps p_i. The latter cannot happen since p_i has no overlaps. So by contradiction it is proven that if L1 applies for some point p, our algorithm will label it.

Rule L2
L2 applies if some p has a candidate p_i that only overlaps some q_k, and q has a candidate q_j $(j \neq k)$ that is only overlapped by p_l $(l \neq i)$, then add p_i and q_j to the solution and eliminate all other candidates of p and q.

Rule L3

Additional guarantees

1-slider model

Summary

The 1-slider algorithm also uses a quadtree and a heap. It firstly initializes this quadtree, filling it with all of the points in the input. Then, each of the points will be assigned a so-called *bounding box*, which represent all potential label placements. These *bounding boxes* will be given collision values by the quadtree. After this, a maxheap of points is initialized, using the amount of collisions the *bounding box* of the point has with other *bounding boxes* as keys. Now the program will enter a loop. In this loop, we find a solution for all of the points which have not been *discarded*. Firstly, we have to re-assign collision values for all of the remaining points. Secondly, we apply rule L1

and L2, which have been explained earlier. Thirdly, we finish the solution by executing a greedy algorithm on the remaining points. Lastly, we check whether the solution we found this iteration is better than the best solution so far. If it is better, we replace the best solution with this new solution. Now we delete the point of which the *bounding box* has the most collisions from the heap and mark the corresponding point in the quadtree as *discarded*. This loop is repeated until the heap is empty, we have found a solution which places labels on all of the points, or when four and a half minutes have passed. In the end, we report the best solution found. A detailed explanation of all subalgorithms and concepts is given below.

Algorithm 6 1slider algorithm

```
1: procedure OneSlider
2:     mostfound = 0
3:     bestsolution = empty array of points
4:     initialize timer
5:     build quadtree containing all of the points
6:     give collisions to all of the bounding boxes on the points
7:     build heap of points with number of potential collisions as keys
8:     while heap not empty AND timer < time limit AND mostfound != amount of
   points in input do
9:         give collisions to the bounding boxes of the undiscarded points
10:        try L1 for each unplaced and undiscarded point
11:        try L2 for each unplaced and undiscarded point
12:        do the greedy part for each unplaced and undiscarded point
13:        do the greedy part for all unplaced points (including discarded)
14:        delete root of the heap and mark corresponding point in quadtree as discarded
15:        if amount of labels placed this iteration > mostfound then
16:            mostfound = amount of labels placed this iteration
17:            bestsolution = solution found this iteration
18:        end if
19:    end while
20:    return bestsolution
21: end procedure
```

Bounding Boxes

As mentioned earlier, *bounding boxes* represent the outer-most bounds for a label when considering all possible placements using the 1-slider model. Every point p is assigned such a *bounding box*. On the y-axis, the starting point of the bounding box will be the y-coordinate p_y of the point. The length of the interval will be the label height, so the interval ends in $p_y + labelheight$. On the x-axis, the interval will start at $p_x - labelwidth$

and end at $p_x + labelwidth$. In this way, the *bounding box* will precisely be the rectangle which represents all possible placements of the label. Every *bounding box* B also has a reference to its original point p and a list of *bounding boxes* it intersects with. This list is created by using the function *GiveCollisions* in the Quadtree, which was explained earlier. Now that we have these lists, we can go through all *bounding boxes* and give each *bounding box* two collision values, x_1 and x_2. x_1 represents the x-coordinate for which the area to the left of x_1 contains collisions with other *bounding boxes* to the left. x_2 represents the x-coordinate for which the area to the right of x_2 contains collisions with other *bounding boxes* to the right.

The heap

Beside the quadtree, the algorithm also uses a maxheap to store the points. The keys in the maxheap are the amount of collisions the *bounding boxes* of the points have with other *bounding boxes*. This way, we can easily remove the point of which the *bounding box* has the most collisions with other *bounding boxes*: we simply extract the root of the maxheap in $\Theta(log(k))$ time, with k being the amount of points left in the heap.

Rule L1

We will now explain how rule L1 is implemented in the 1slider algorithm. Recall that L1 places all labels which can not possibly have any intersections with other labels. We implement this by comparing the found collision values x_1 and x_2 for each *bounding box* B of each point p. If there is enough space in between x_1 and x_2 to place a label, so if $x_2 - x_1 \geq labelwidth$, then we can be sure that we can place a label there without it ever intersecting any other possible label placement. This is because x_1 is the right-most position for which there are collisions to the left with *bounding boxes* to the left and x_2 is the left-most position for which there are collisions to the right with *bounding boxes* to the right. This means that there are no collisions to the right of x_1 until we reach x_2 and no collisions to the left of x_2 until we reach x_1. Therefore, there are no possible overlaps in between x_1 and x_2 and we can place a label there which satisfies L1. We place the label on p with slider value $\frac{p_x - B_{x_1}}{labelwidth} + 1$. This places it with the left bound of the label at x-position B_{x_1}. Since $x_2 - x_1 \geq labelwidth$, the right bound of the label will be $x_1 + labelwidth \leq x_2$, so the label will be in between x_1 and x_2.

Rule L2

After L1 is applied, rule L2 is executed. Recall that L2 places labels on two points which have candidates which only collide with each other. To do this, we search for points with *bounding boxes* which have only one collision. We then check whether the

bounding boxes they collide with also have only one collision. If that is the case, we can say that both *bounding boxes* only have one collision and we have a case of L2. We call the left-most point in this situation p_1 and the right-most point p_2. If the points have equal x-positions, this is assigned randomly. For each of these cases, we give p_1 slider value 0 and the p_2 slider value 1. This way, p_1 gets a label which is placed all the way to the left and the p_2 gets a label which is placed all the way to the right. The two labels will never intersect any labels. They will not intersect each other, because the label on p_1 will occupy the x-interval $[p_{1_x} - labelwidth,\ p_{1_x}]$ and the label on p_2 will occupy the x-interval $[p_{2_x}, p_{2_x} + labelwidth]$. Since $p_{1_x} \leq p_{2_x}$, the labels will never intersect each other. The labels will also never intersect labels outside of the L2 case, because the *bounding boxes* of these points only collide with each other, so they can not intersect other potential label placements.

Greedy part

After L1 and L2, we complete the solution by executing a greedy part on the remaining points. Firstly, we do this for all undiscarded points and then we see if there are any discarded points on which we can still place a label.

Reason for deleting the point with most collisions

Running time

The running time of this algorithm is dependent on the amount of points n in the input and the total amount of collisions c the *bounding boxes* have. For example, if we find no collisions at all, we have n cases of L1 and the algorithm terminates after the first iteration of the main loop, reporting a solution which places labels on all points. Therefore, the running time of the algorithm will be described in terms of n and c.
Initializing the quadtree simply adds every point in the input to the quadtree. Adding a point to the quatree takes $\Theta(log(n))$ time, so adding every point will take $\Theta(n * log(n))$ time. Initializing the heap with all of the points takes $\Theta(n)$ time. The time it takes to give collisions to the *bounding boxes* will depend on the amount of points, the size of the labels, and the point distribution.

Now the algorithm enters the loop. This loop is repeated until the heap is empty, a solution which places labels on all of the points is found, or four and a half minutes have passed. It is nearly impossible to predict when our algorithm will find a solution which places labels on all of the points, so we will not consider this in the running time. Therefore, the total running time of the loop will be the minimum of the time it takes to empty the heap and 4.5 minutes. Firstly, we will determine the time it takes to empty the heap. At the first iteration, the heap contains n points. At the end of each iteration, one point is removed from the heap. Therefore, the loop is repeated n times. Inside the

loop, the running time depends on the running time of applying L1, applying L2, and executing the greedy part.