Stochastic U-Curve Branch and Bound

Student: Gustavo Estrela de Matos

Advisor: Dr. Ulisses Braga-Neto

1 Input Generation

To generate the input for the problem we created two functions. The first creates a vector of floating points that simulates the values of a chain of a boolean lattice that respects the U-Curve assumption; and the second one adds a random noise to the values of the vector.

1.1 Chain Generation

The chain is generated based on a polynomial of second degree with positive leading coefficient. This polynomial has the minimum value of 0.1 evaluated at some point $x^* \in [0, 1]$ which is determined by the user. The algorithm receives two arguments: the first one, n, represents the size of the chain; and the second one, $center \in [0, 1]$, specifies where is the minimum of the polynom. After creating a random polynom that holds our constraints, we split the $[0, 1]$ interval in n equaly spaced points, and evaluate the polynomial at these points, generating the curve.

1.2 Noise

The noise is applied to the vector created by $GeneratePoints$, by adding that is in the interval $[-\alpha\frac{curve\_amplitude}{n}, \alpha\frac{curve\_amplitude}{n}]$, where $curve\_amplitude = max(v) - min(v)$ and α is a random variable with gaussian distribution with mean 0 and standard deviation σ.

Algorithm 1 U-Curve Input Creator

procedure GENERATEPOINTS($n, max\_distance, center$)
  $points \leftarrow \{0, ..., 0\}$
  $p.a \leftarrow random()$
  $p.b \leftarrow -2 * p.a * center$
  $p.c = (p.a^2 - 0.1)/(4 * p.a)$
  $dx \leftarrow 1/n$
  $x \leftarrow 0$
  for $i \in \{0, ..., n\}$ **do**
    $points_i \leftarrow p(x)$
    $x \leftarrow x + dx$
  end for
  $j \leftarrow n * random()$
  $plain\_size \leftarrow (n - j) * random()$
  for $k \in \{1, ..., plain\_size\}$ **do** ▷ Creates a plain area in the chain
    $points[j + k] \leftarrow points[j]$
  end for
  return $points$
end procedure

Figure 1: Example of a curve generated with $\sigma = 0$

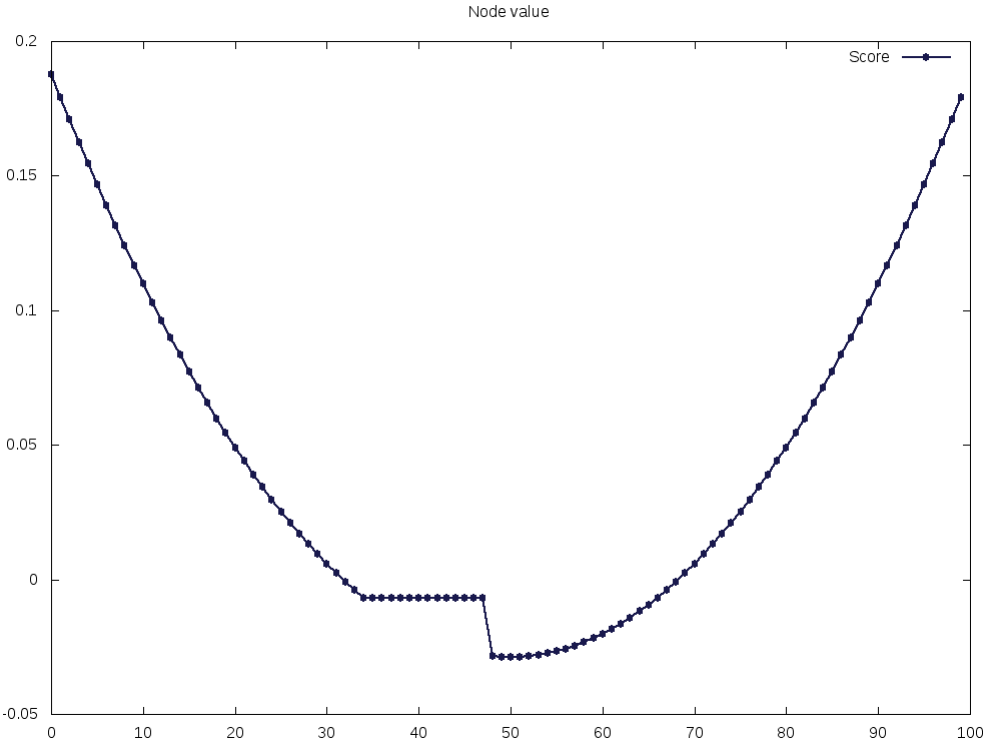


Figure 2: Example of a curve generated with $\sigma = 1$

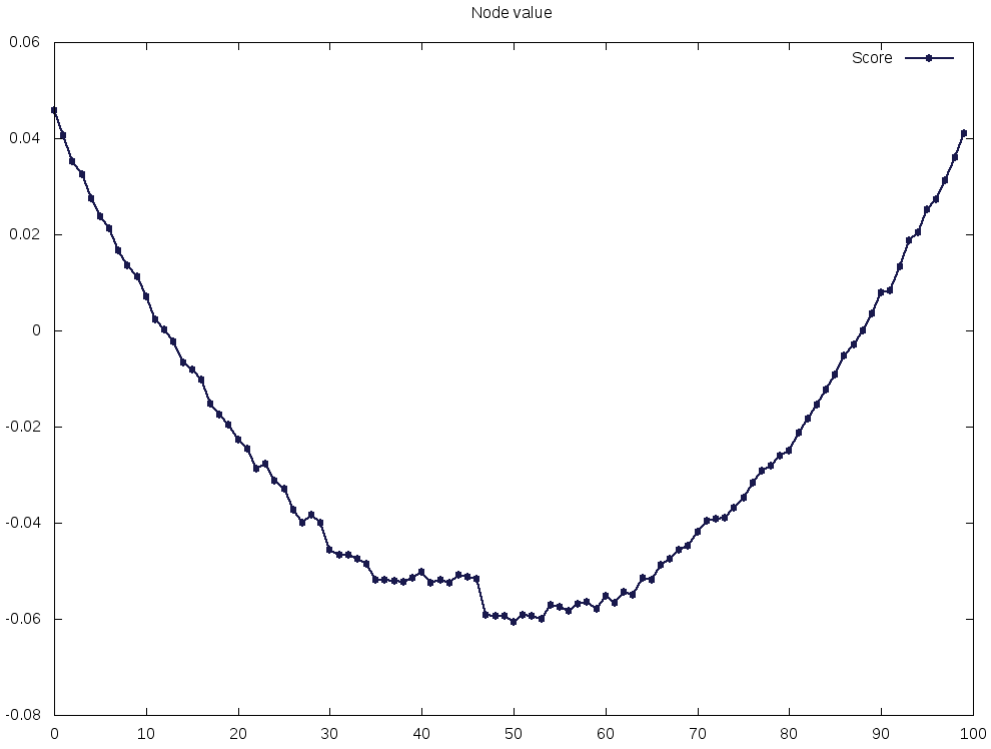
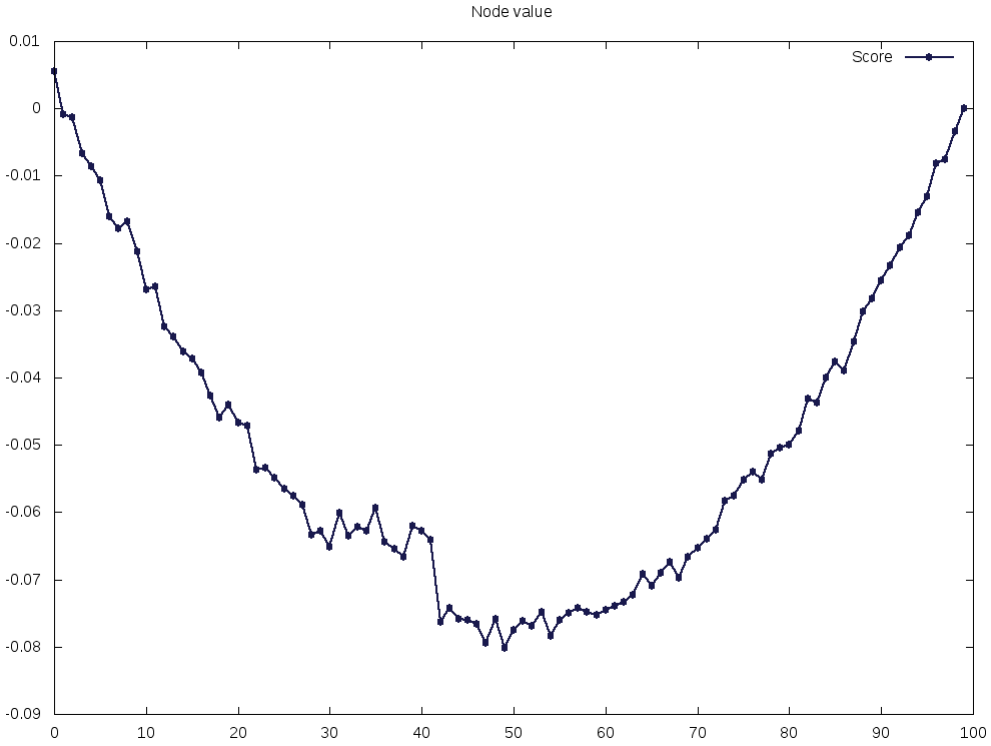


Figure 3: Example of a curve generated with $\sigma = 2$



2 Bisection Algorithms

2.1 Traditional Bisection

This is the simplest bisection algorithm we implemented. The basic idea of this algorithm is to divide the original problem in two halfs and, by analysing the neighbours of the node in the middle, determine which half has the minimum and then solve this half recursively. To guarantee the optimally of the algorithm (when the input respects the U-Curve assumption) we solve both halfs when the neighbours can't determine if the minimum lies to the left or right of the middle point.

Algorithm 2

```
1: procedure BISECTION(v)
2:     n ← v.length
3:     i ← n/2
4:     if (valley(v, i)) then
5:         return v[i]
6:     else
7:         direction ← SelectSide(v, i)
8:         if direction = Left then
9:             return Bisection([v_i, ..., v_n])
10:        else if direction = Right then
11:            return Bisection([v_0, ..., v_{i-1}])
12:        else                                          ▷ Unknown direction
13:            return min(Bisection([v_0, ..., v_{i-1}]), Bisection([v_i, ..., v_n]))
14:        end if
15:    end if
16: end procedure
```

Algorithm 3

```
1: procedure SELECTSIDE(v, i)
2:     d = v[i + 1] − v[i − 1]
3:     if |d| < ε then
4:         return Unknown
5:     else if d > 0 then
6:         return Left
7:     else
8:         return Right
9:     end if
10: end procedure
```

2.2 Mid-neighbour Bisection

As an attempt to minimize the effects of noise in the traditional bisection algorithm we developed the Mid-neighbour Bisection, which brings two new ideas to the traditional bisection, that changes the evaluated points and also how we divide the problem in smaller problems.

The first idea consists in changing the points that are evaluated to decide which side to go. Instead of looking to the neighbours of the middle point, we now evaluate the points that are in the middle of the left half and right half. After evaluating these points, we calculate the difference between the left-middle point and middle point and also the difference between the right-middle point and the middle point to get two abstract slopes that can guarantee us which fraction of the input has the minimum value.

The second idea considers the "reliabillity" of the abstract slopes mentioned before, for instance, if we say that the left slope (middle point - left-middle point) is negative, it's more likely that the function is going to increase from the left-middle point and before if the distance between these two points is greater. That happens because if the function decreases from the left-middle point and before, then the left-middle point suffered some noise that caused a peak (an inverted u-shape) and, considering that the points should describe a u-shape that would imply in a bigger noise for bigger distances of middle point and lef-middle point. We modeled the "reliability" of the abstract slope as a linear function that decreases from one to zero in lgn steps.

The abstracts slopes are reduced to three different cases: 1 for growing, -1 for decreasing and 0 for plain areas or unreliable difference. Since we have three different slopes, there are nine different cases that need to be handled by the algorithm. Suppose that we are using the mid-neighbour bisection on a vector $v = \{v_1, ..., v_n\}$ and the right-middle, middle and left-middle point are, respectively, lm, m, and rm. Then, the following pictures represent how the algorithm finds the minimum recursively.

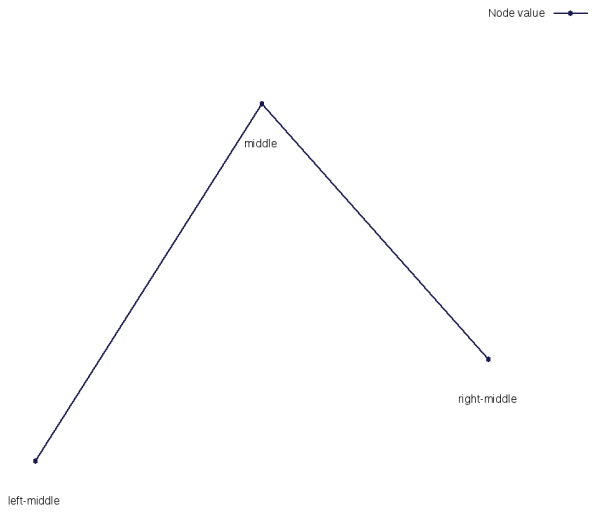


Figure 4: In this case we split the original problem into the two problem of finding the minimum of $\{v_1, ..., v_{m-1}\}$ and $\{v_m, ..., v_n\}$.

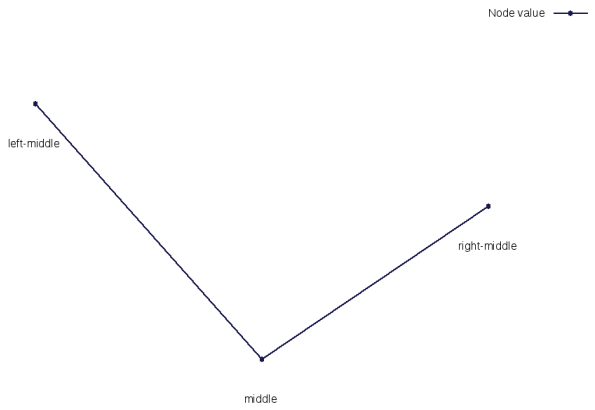


Figure 5: In this case we reduce the problem to finding the minimum of $\{v_{lm}, ..., v_{rm}\}$.

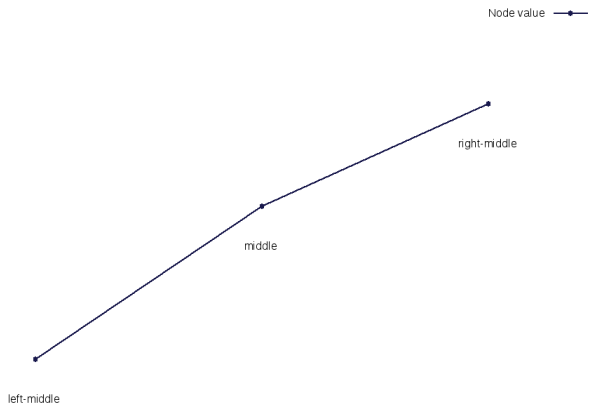


Figure 6: For this case we reduce the problem to finding the minimum of $\{v_1, ..., v_m\}$.

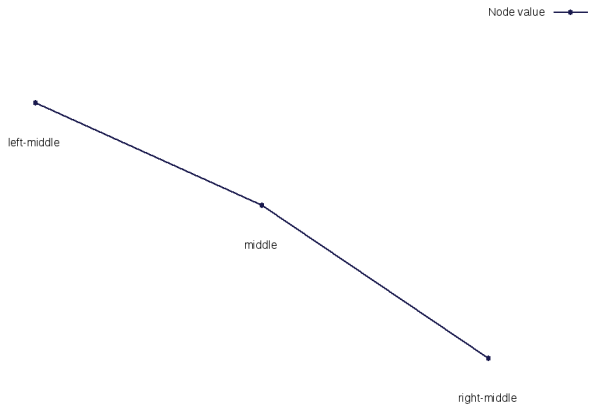


Figure 7: In this case we reduce the problem to finding the minimum of $\{v_m, ...v_n\}$.

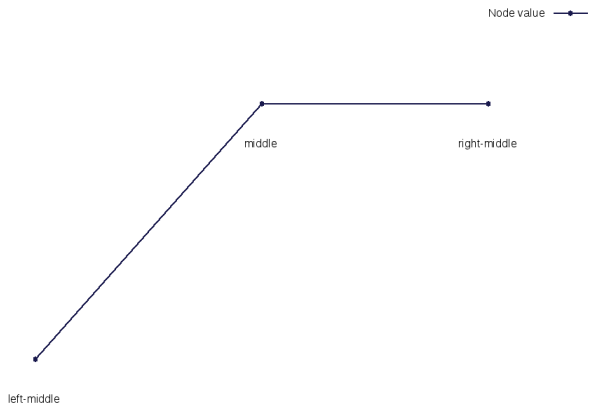


Figure 8: For this case we reduce the problem to finding the minimum of $\{v_1, ..., v_m\}$.

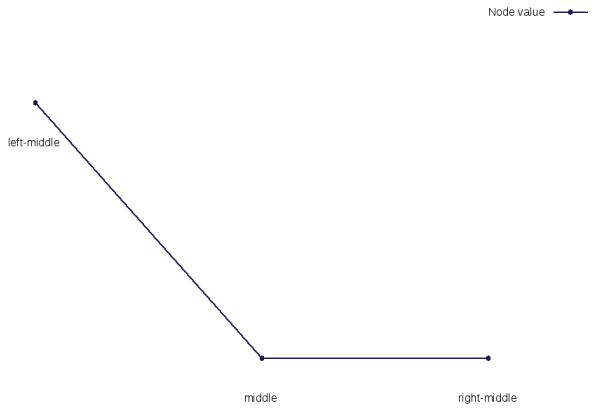


Figure 9: For this case we reduce the problem to finding the minimum of $\{v_{lm}, ..., v_m\}$.

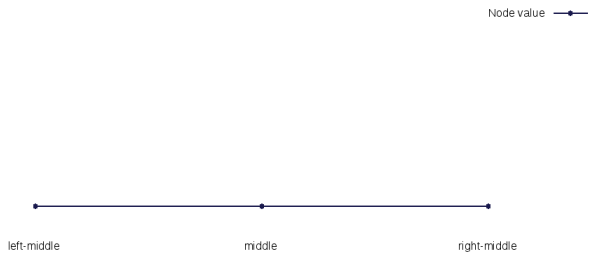


Figure 10: In this case we split the original problem into the two problem of finding the minimum of $\{v_1, ..., v_{m-1}\}$ and $\{v_m, ..., v_n\}$.

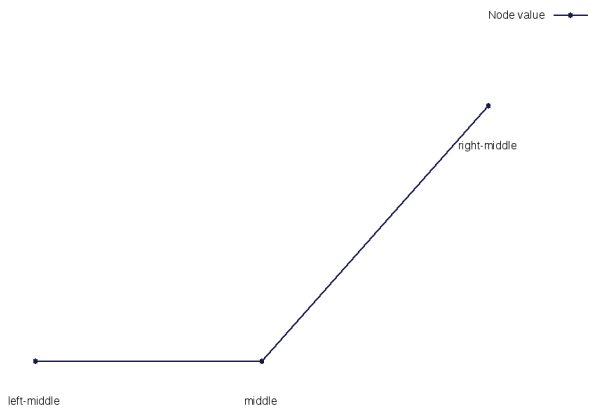


Figure 11: In this case we reduce the problem to finding the minimum of $\{v_1, ..., v_m\}$.

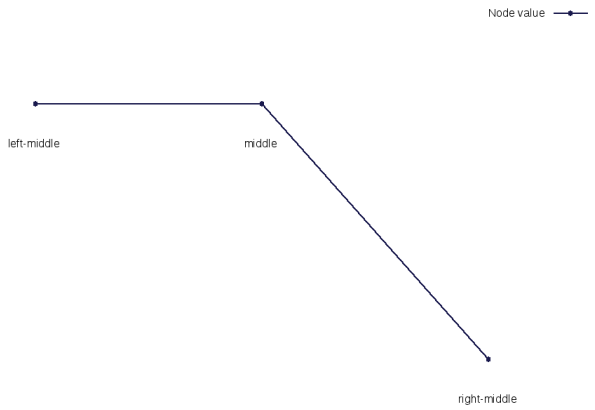


Figure 12: In this case we reduce the original problem to finding the minimum of $\{v_m, ..., v_n\}$

2.3 U-Curve Probabilistic Bisection

The U-Curve Probabilistic Bisection (UPB) is a stochastic version of the algorithm we called "Traditional Bisection". The UPB algorithm keeps a probability mass function $f_i(x)$, for every iteration i and every node x of the search space, that can represent the algorithm belief, on iteration i, that the node x is the answer to the problem. This algorithm has two key points: selecting the node to be evaluated and updating the probability mass function.

Different from the Traditional Bisection, we don't select the element that is the middle of the input to evaluate, instead we select the m-th element of the input vector v such that:

$$m = \min_{1 \leq i \leq n} \{i \mid f(v_i) \geq \frac{1}{2}\}.$$

, i.e the median of the possible solutions. The algorithm then compares the adjacent nodes of v_m and decides which side the solution should be the same way it was done in the traditional bisection.

We call p_c the probability that we chose the correct side when evaluating v_m's neighbours and $\alpha = \sum_{j=1}^{m} f_i(v_j)$. In the case where the actual solution lies in the right side of v_m, the

probability mass function update is calculated by:

$$f_{i+1}(y) = \begin{cases} (\frac{1}{1-\alpha})p_c f_i(y) \text{ if } y \geq v_m \\ \alpha(1-p_c)f_i(y) \text{ if } y < v_m \end{cases}$$

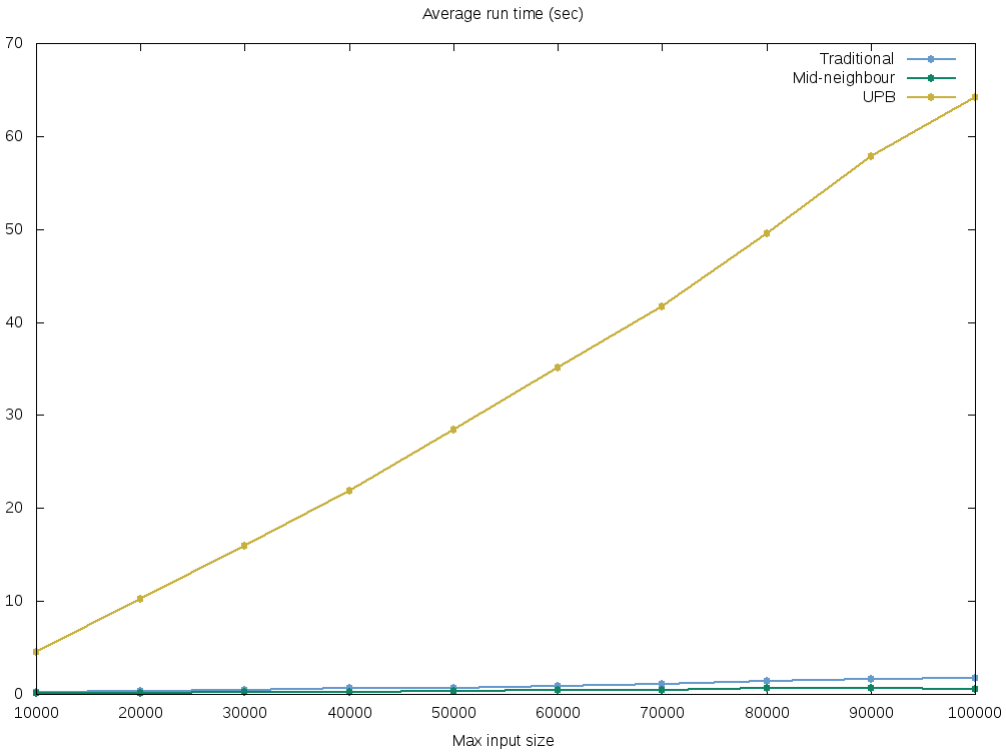
. This update is similar for the case when the solution lies in the left side of v_m. If there is no defined value for f_0, we simply assume that the probability of being a solution is distributed uniformily between all nodes of the chain.

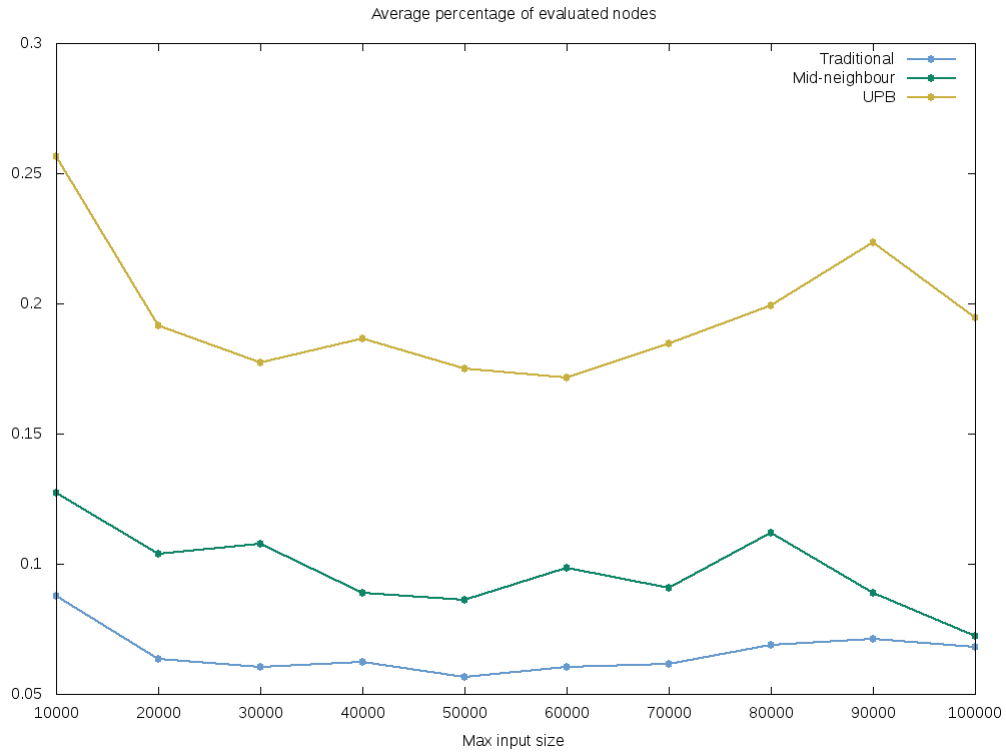
The algorithm terminates when the first, second (median) and third quartile of the possible solution lies exactly in the same node, the solution given by the algorithm. This termination condition avoids an early termination, that would be caused by a local minimal element if we used the same termination condition as we did in the traditional bisection.

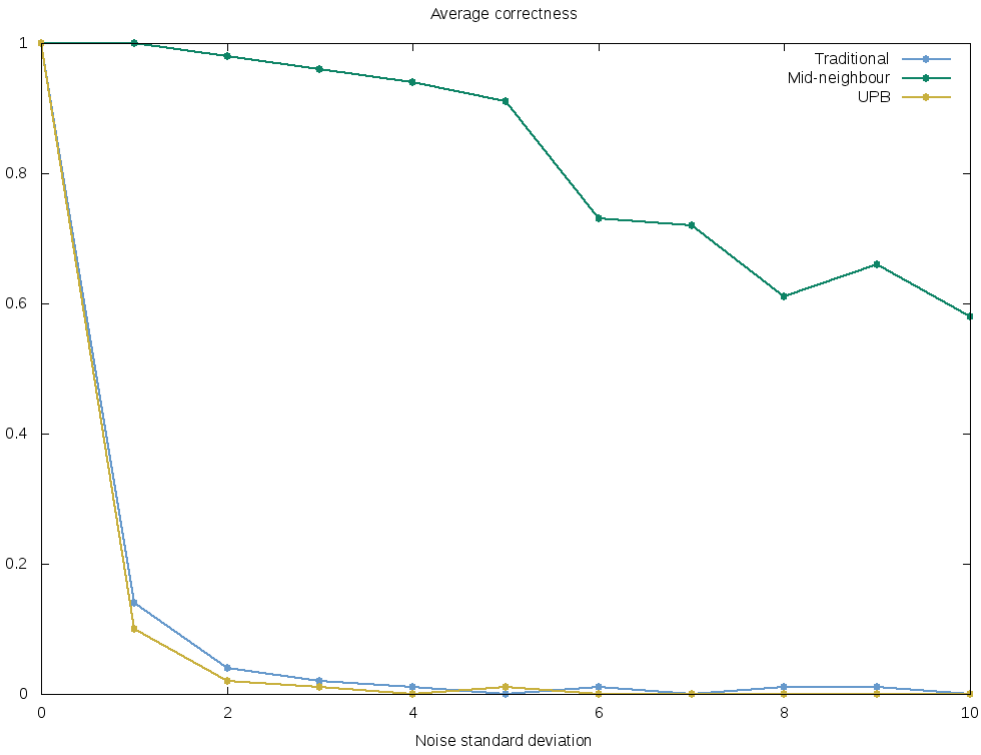
Algorithm 4

```
1: procedure UPB(v, pmf)
2:     n ← v.length
3:     Quartiles ← FindQuartiles(pmf)
4:     while Quartiles_1 ≠ Quartiles_2 and Quartiles_2 ≠ Quartiles_3 do
5:         direction ← SelectSide(v, i)
6:         if direction is Unknown then                    ▷ Unknown direction
7:             return SplitUPB(v, pmf, i)
8:         else
9:             UpdatePMF(pmf, i, alpha, direction)
10:        end if
11:        i ← FindMedian(pmf)
12:        α ← F(v_i)
13:    end while
14:    return v_i
15: end procedure
```

3 Results


Average run time (sec)
Traditional
Mid-neighbour
UPB
70
60
50
40
30
20
10
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Max input size


Average percentage of evaluated nodes
Traditional
Mid-neighbour
UPB
0.3
0.25
0.2
0.15
0.1
0.05
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Max input size


Average correctness
Traditional
Mid-neighbour
UPB
1
0.8
0.6
0.4
0.2
0
0
2
4
6
8
10
Noise standard deviation