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 algorithm receives three parameters: n, $max\_distance$ and $center$; and returns as output a vector that has values from 0 to 1 and respects the U-Curve assumption. The first parameter defines the size of the chain; the second represents the greatest possible difference between the values of neighbour nodes, which is a random value uniformily distributed between 0 and $max\_distance$; and the last represents the index of the node with minimum value.

1.2 Noise

The noise is applied to the vector created by $GeneratePoints$, by adding a value uniformily distributed in the interval $[-\alpha\frac{curve\_amplitude}{n}, \alpha\frac{curve\_amplitude}{n}]$, where $curve\_amplitude = max(v) - min(v)$ and α is a noise parameter.

Algorithm 1 U-Curve Input Creator

procedure GENERATEPOINTS($n, max\_distance, center$)
    $points \leftarrow \{0, ..., 0\}$
    $minimum \leftarrow \frac{random()}{2}$
    $points[center] \leftarrow minimum$

    for $i \in \{0, ..., center - 1\}$ **do**
        $points[i] \leftarrow points[i + 1] + (1 - points[i + 1]) * random()$
    end for
    for $i \in \{center + 1, ..., n - 1\}$ **do**
        $points[i] \leftarrow points[i - 1] + (1 - points[i - 1]) * random()$
    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 forreturn $points$
end procedure

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

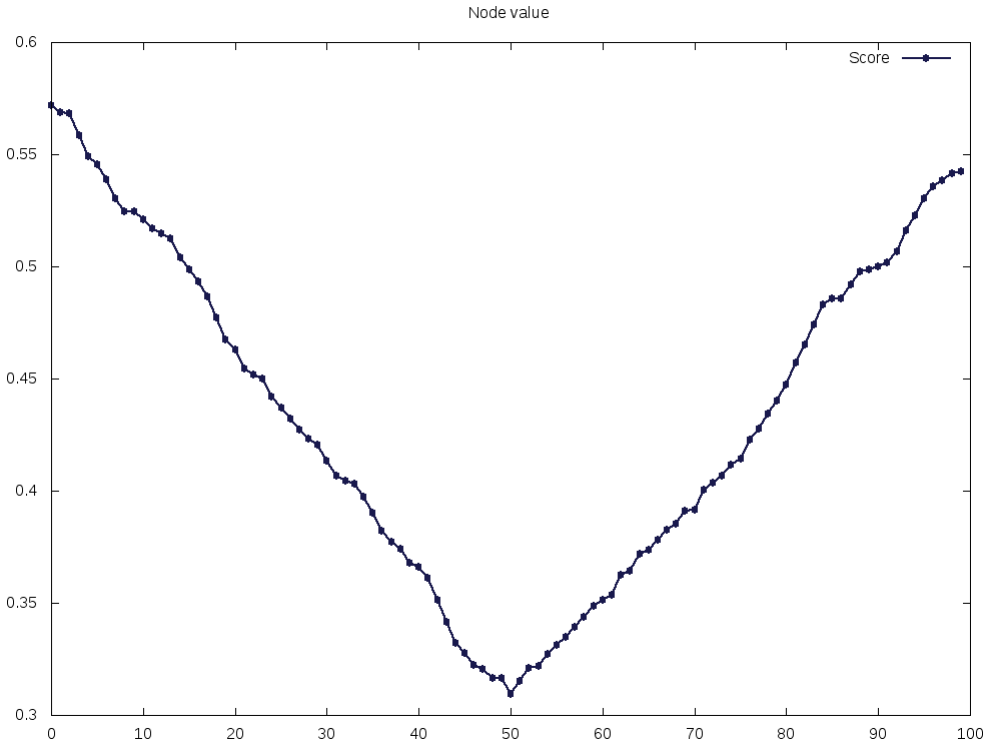


Figure 2: Example of a curve generated with $\alpha = 1$

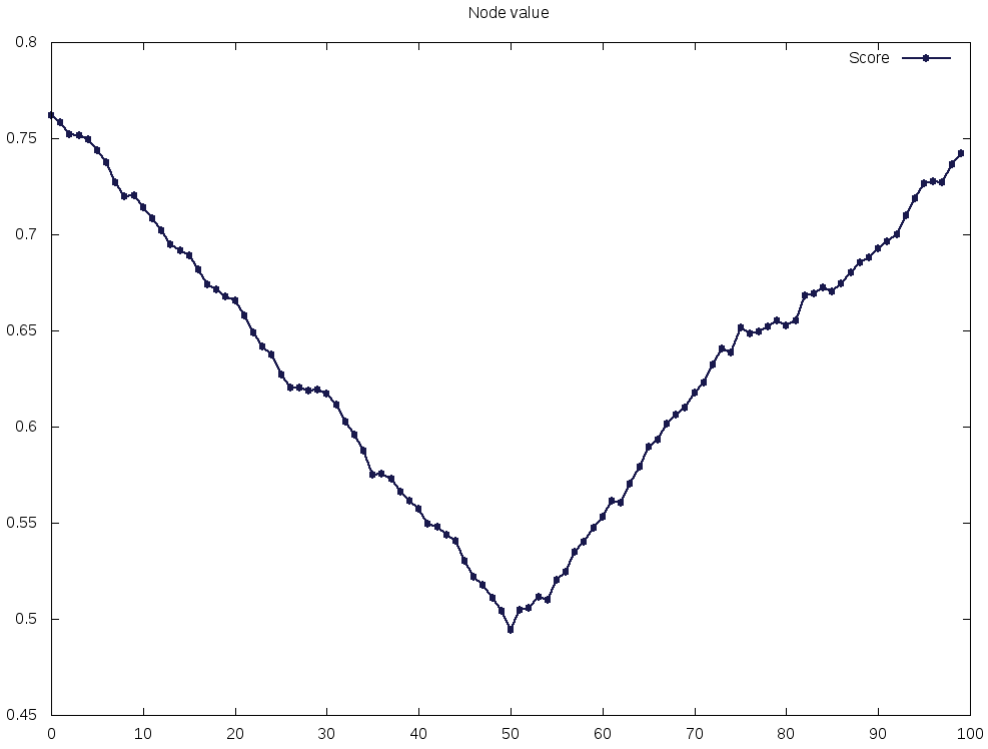
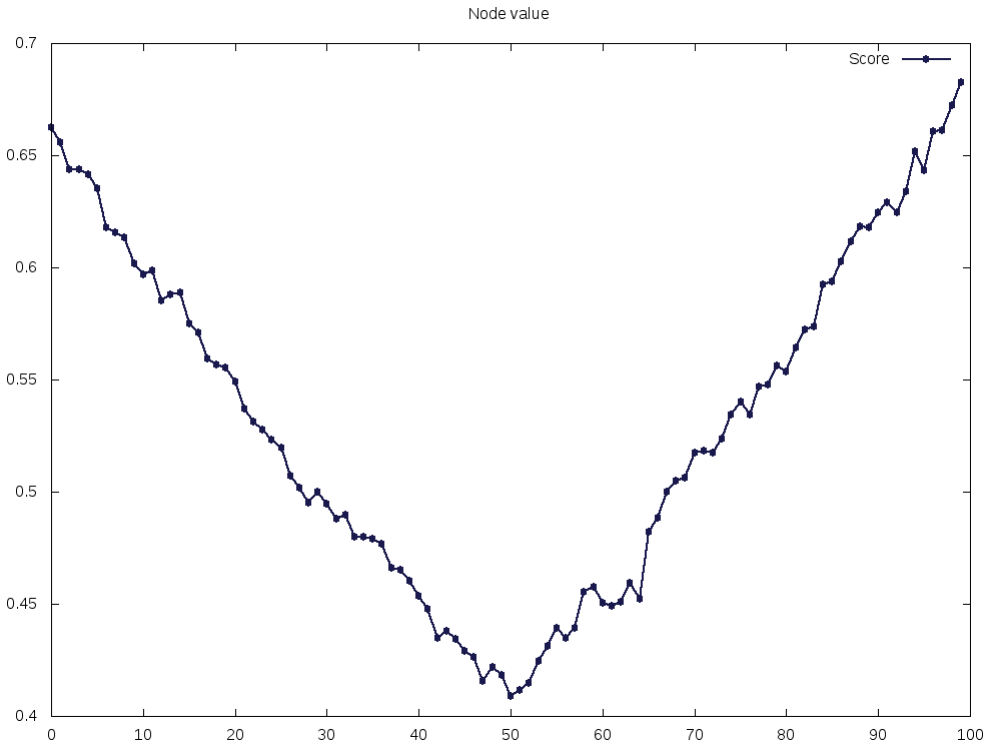


Figure 3: Example of a curve generated with $\alpha = 2$



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 handled by the algorithm, which we describe in the following pictures.

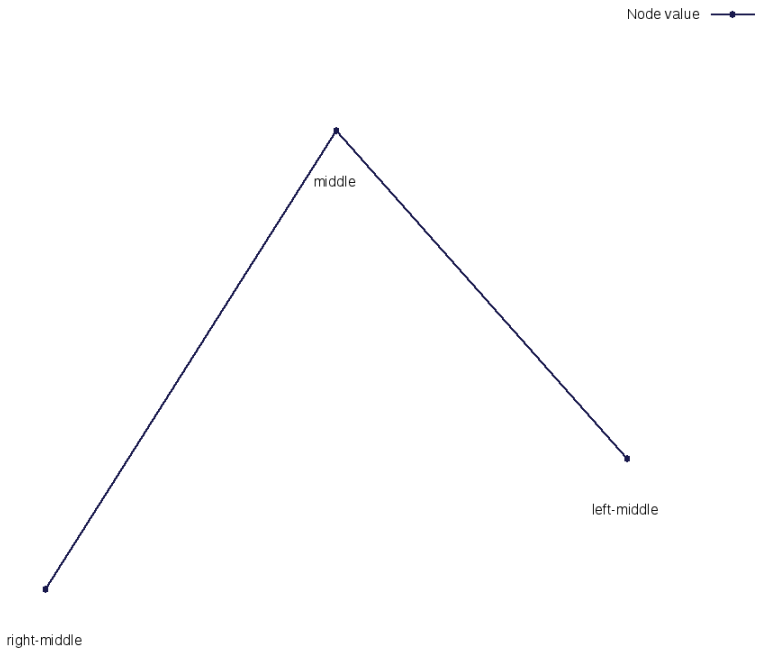


Figure 4: Caption

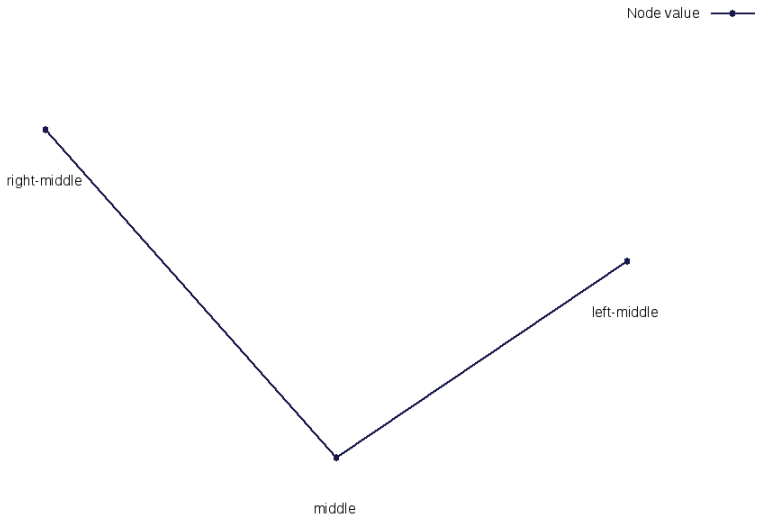


Figure 5: Caption

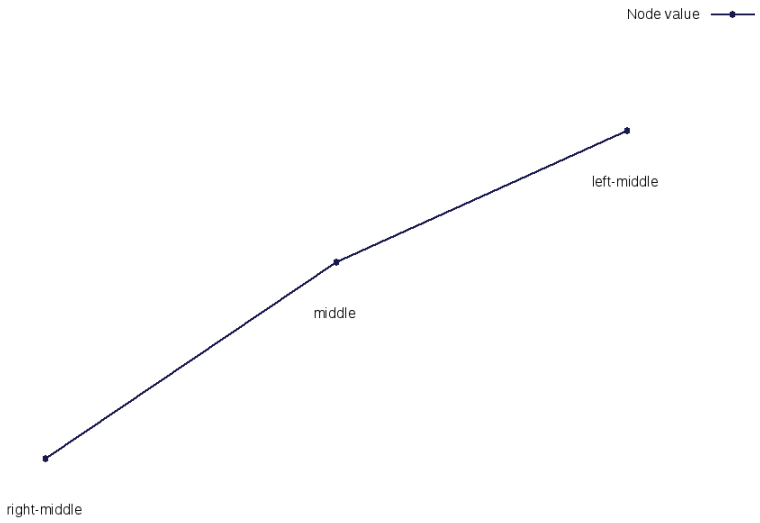


Figure 6: Caption

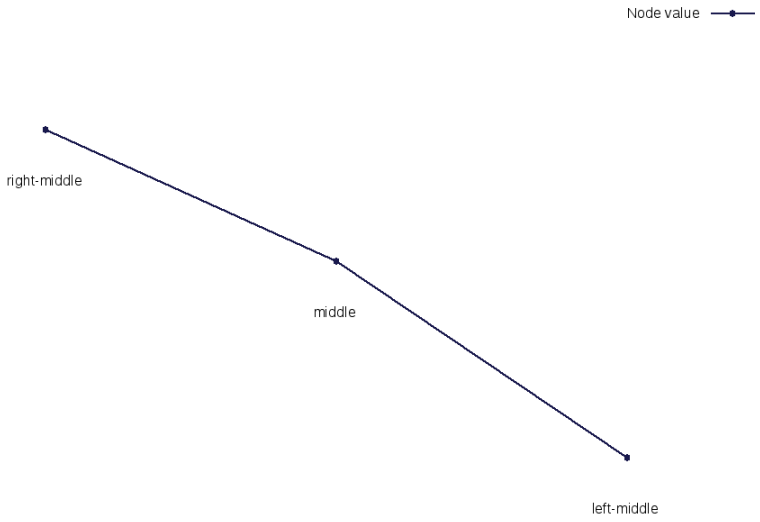


Figure 7: Caption

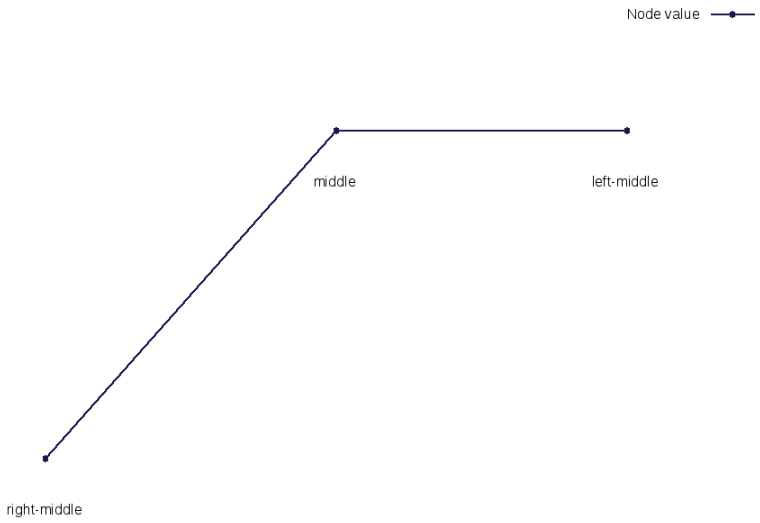


Figure 8: Caption

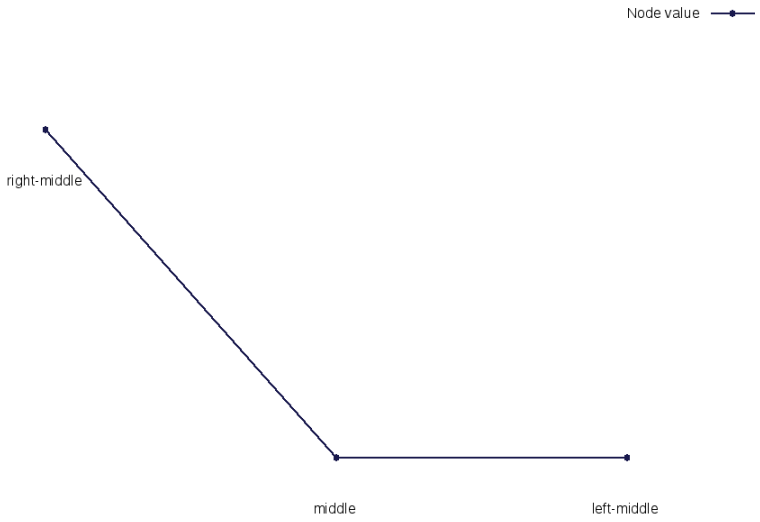


Figure 9: Caption

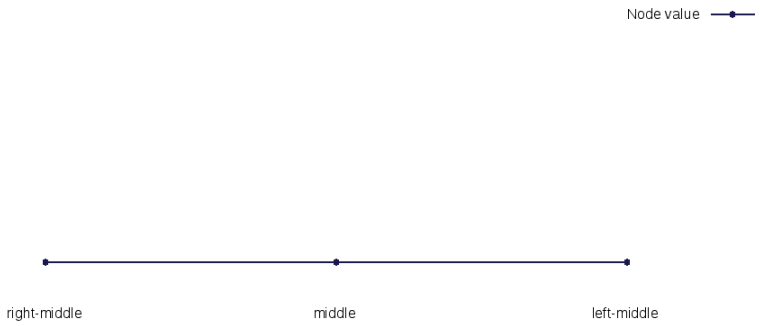


Figure 10: Caption

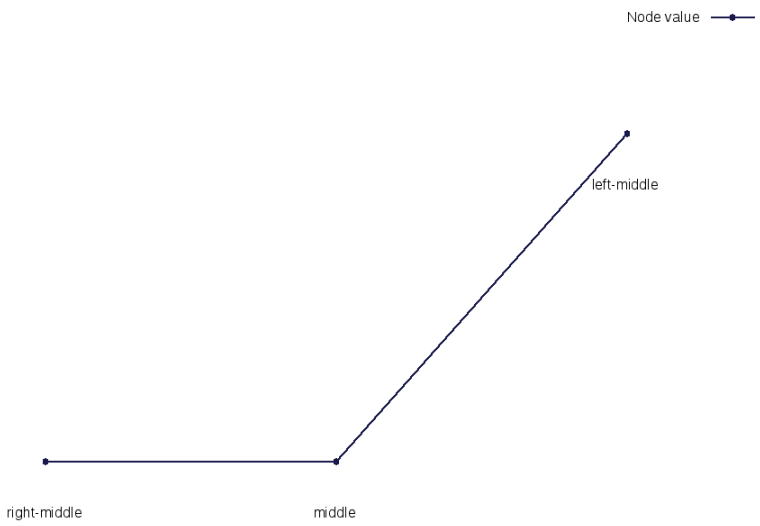


Figure 11: Caption

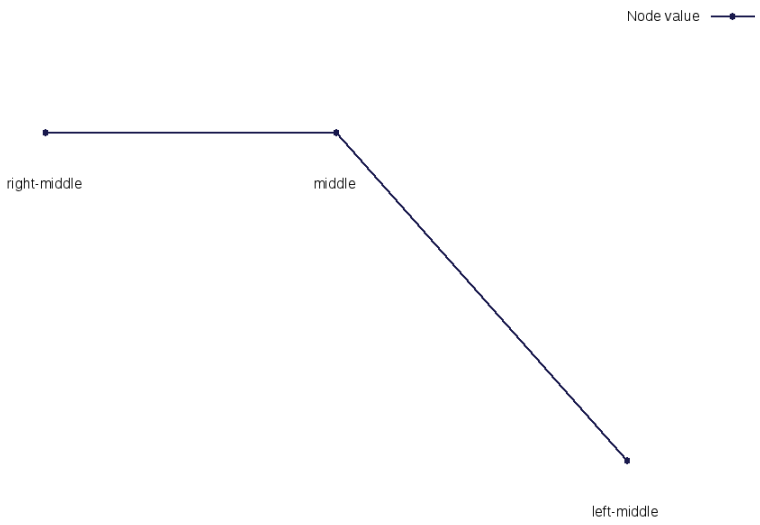


Figure 12: Caption