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 $\alpha = 0$

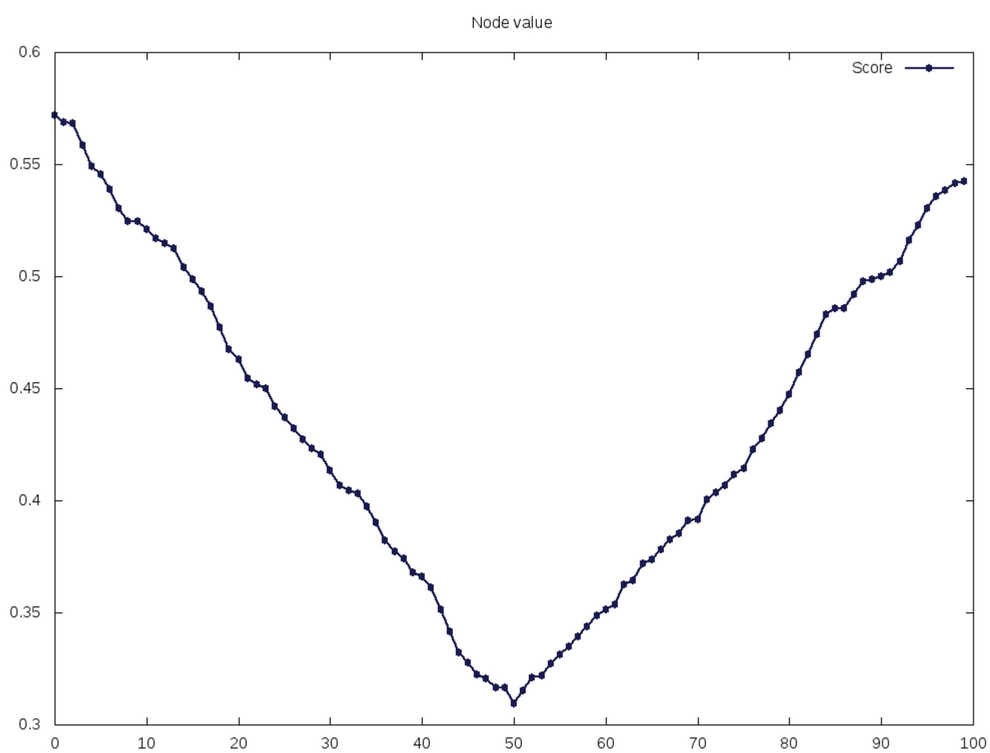


Figure 2: Example of a curve generated with $\alpha = 1$

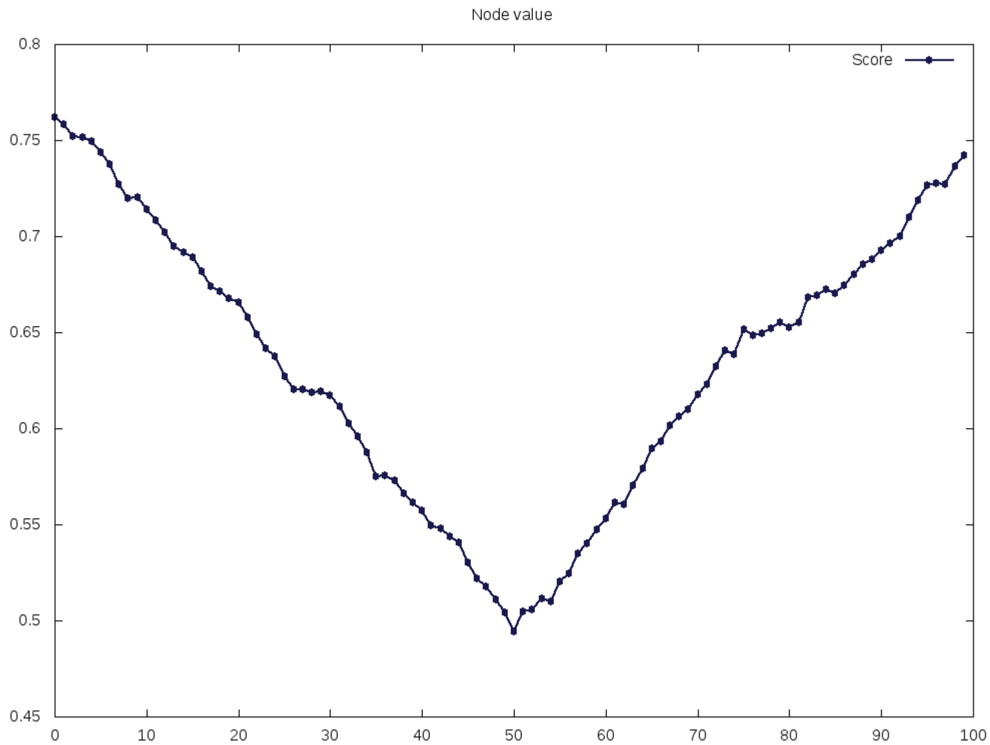
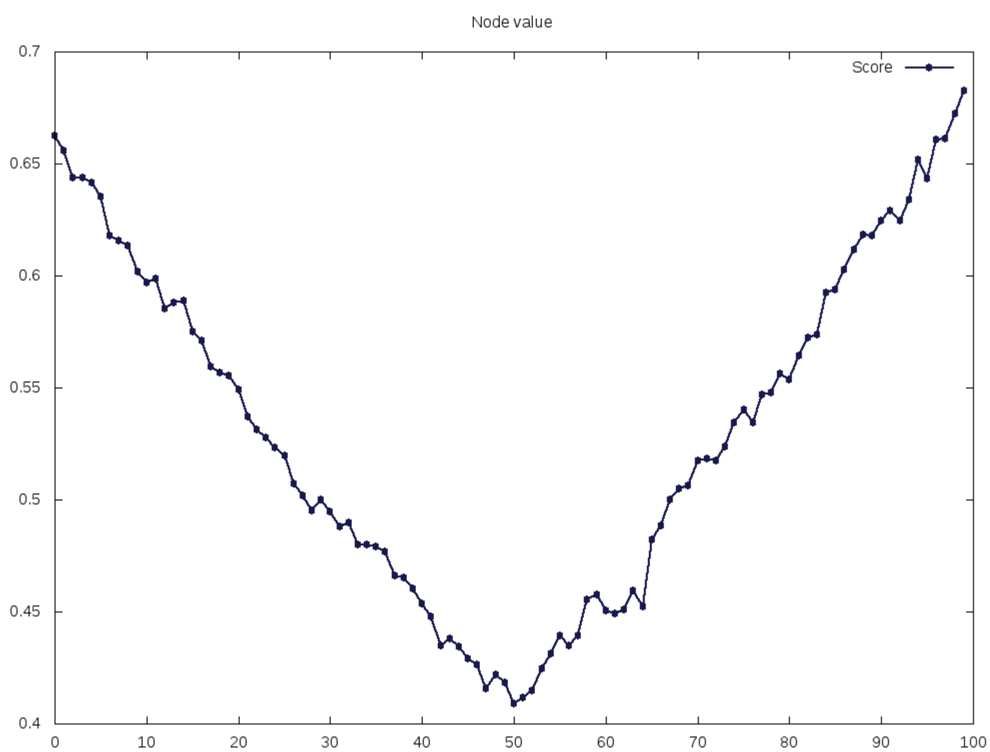


Figure 3: Example of a curve generated with $\alpha = 2$



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

```
procedure BISECTION(v)
    n ← v.length
    i ← n/2
    if (valley(v, i)) then
        return v[i]
    else
        direction ← SelectSide(v, i)
        if direction = Left then
            return Bisection([v_i, ..., v_n])
        else if direction = Right then
            return Bisection([v_0, ..., v_{i-1}])
        else                                        ▷ Unknown direction
            return min(Bisection([v_0, ..., v_{i-1}]), Bisection([v_i, ..., v_n]))
        end if
    end if
end procedure
```

Algorithm 3

```
procedure SELECTSIDE(v, i)
    d = v[i + 1] − v[i − 1]
    if |d| < ϵ then
        return Unknown
    else if d > 0 then
        return Left
    else
        return Right
    end if
end procedure
```