Data structures and Algorithms (INFO-F413)
Assignment 1: Binary Space Partitions

Hakim Boulahya
hboulahy@ulb.ac.be

Université Libre de Bruxelles

November 8, 2018

Contents

1 Implementation

1.1 Solution for the subproblems

1.1.1 Intersection

To find the intersection point between the line that is going through a segment and another segment we resolve a system of equations of the two linear equations representing the line of the segments. Let $a_1x + b_1y + c_1$ and $a_2x + b_2y + c_2$ two lines, we know that the intersection point of the two lines, if it exists, is a point that is on both lines i.e. the results of the following system:

$$\begin{cases} a_1x + b_1y + c_1 = 0 \\ a_2x + b_2y + c_2 = 0 \end{cases} \tag{1}$$

Because we represent a segment with two coordinates (x_1, y_1) and (x_2, y_2), we have to find the equation of the line going through a segment using those coordinates to get the intersection point. We will base our solution on the point-slope formula:

Slope Let (x_1, y_1) and (x_2, y_2) the pair of coordinates corresponding to a segment, the slope of this segment is the value m such as:

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{2}$$

By using one point on the line, i.e. a segment coordinate, and replacing the second point of the point-slope formula by the variables (x, y) we can deduce the coefficients of the line equation as follow:

$$m = \frac{y - y_1}{x - x_1} \iff -mx + y + (mx_1 - y_1) = 0 \tag{3}$$

We can now get the intersection point of a line going through a segment and another segment by using the results explained above. First we calculate slopes of both segments, says m_1 and m_2. If $m_1 = m_2$ the segments have the same slopes therefore they are parallele meaning that there is no intersection. If the slopes are different, we get the intersection of the two lines going through both segments by solving the system of equations in (1). Because we need the intersection of a line and a segment and not two lines, we have to verify that the second segment is indeed cut by the line of the first segment. For this we use the outer product, described in the next section, to check if the coordinates of the second segment are on different side of the line. If coordinates of the second segment are on different sides, in means that the line cuts off the segment.

1.1.2 Split space

For checking on which side of a line a segment is, we use the outer product between the line and both point of the segment that we want to check. To check if the side of a point (x, y) regarding a line that is going through a segment (x_1, y_1), (x_2, y_2) we first evaluate the outer product d:

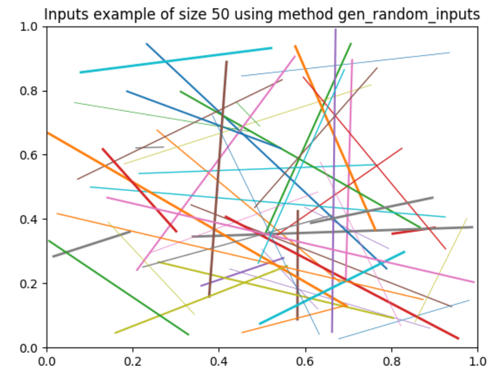
$$d = (x - x_1)(y_2 - y_1) - (y - y_1)(x_2 - x_1)$$

if $d > 0$ the point is on one side, otherwise it is on the other side. With this formula, we can define the side of a segment by checking if both coordinates are one the same side side. If both coordinates are one different sides it means that the line cuts the segment.

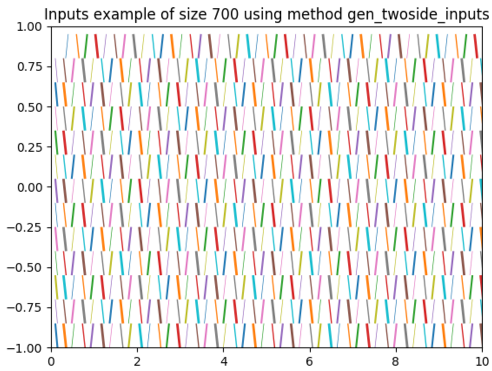
1.2 Inputs

For this experiment we propose three methods of inputs generation:

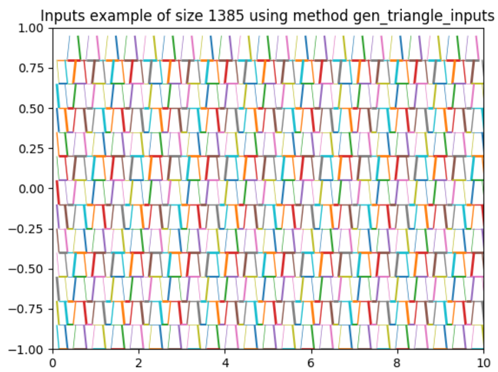
- Completely random inputs. No pattern will be followed when generating those inputs, and both coordinates of all segments will be completely random.
- Inputs following a *triangle*-like pattern, meaning that each pattern will be composed of two side of a triangle, without the base.
- Inputs with all sides of a *triangle*-like pattern.



(a) Completely random inputs



(b) Inputs following a *triangle* pattern with two segments (without the base).



(c) Inputs following a *triangle* pattern with three segments (with the base).

Figure 1: Type of generated inputs

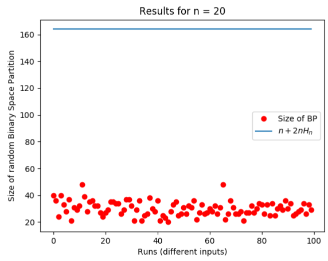
1.3 Algorithm

To get the size of the binary partition size, we implemented the algorithm as described in the assignment description, using the methods describe above for the inputs and the subproblems:

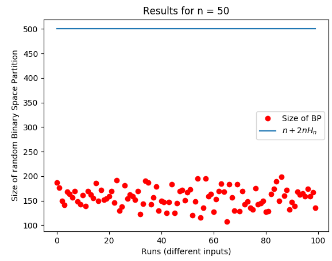
- Generate a list of segments
- Shuffle this list
- Take first element and split in two sides
- Run recursively the algorithm on both sides

2 Results

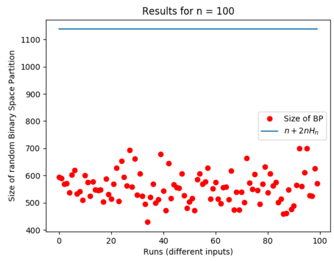
This section shows figures of results of the above algorithms. Each figure contains different plots showing the binary space size (red dots) for a given size and input generation method. The upper bound is shown in a blue line.



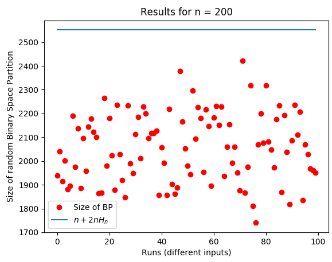
(a) $n = 20$



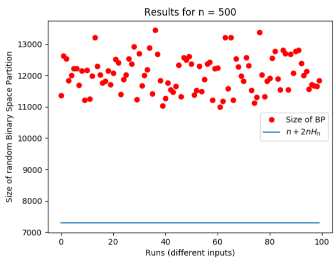
(b) $n = 50$



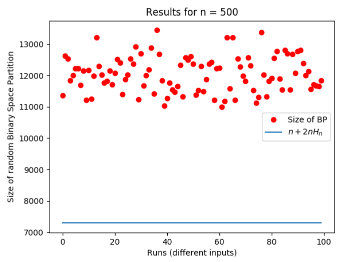
(c) $n = 100$



(d) $n = 200$

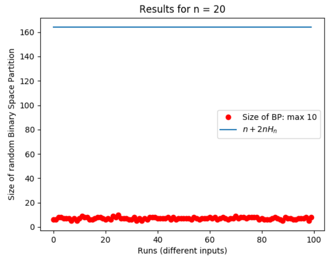


(e) $n = 500$

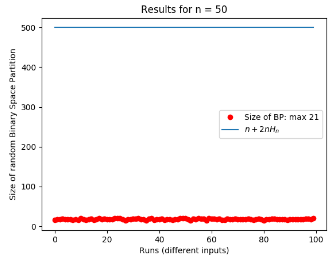


(f) $n = 700$

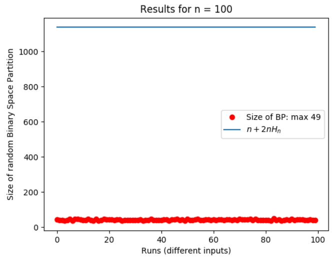
Figure 2: Visualization of a the binary space partition size using random inputs as shown in Figure 1a. Those plots show results for different values of n with 100 expirements per value. Red points represent the BP size and the blue line the upper bound $n + 2nH_n$.



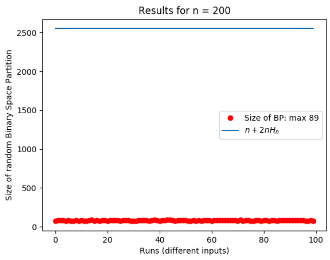
(a) $n = 20$



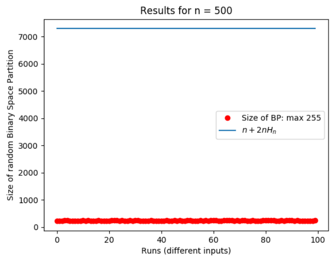
(b) $n = 50$



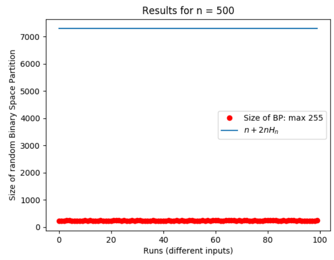
(c) $n = 100$



(d) $n = 200$

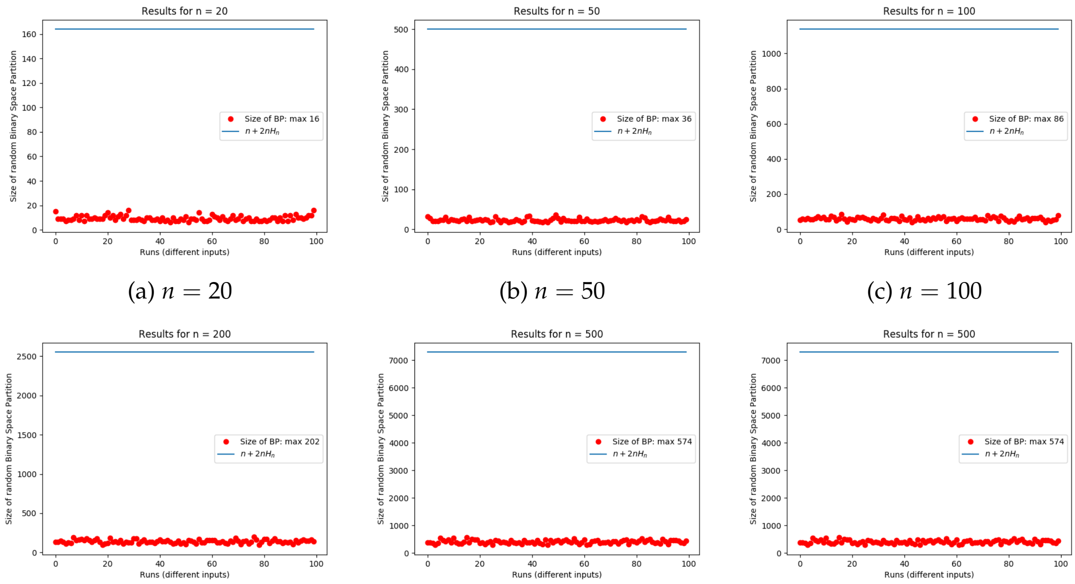


(e) $n = 500$



(f) $n = 700$

Figure 3: Visualization of a the binary space partition size using second method of inputs generation as shown in Figure 1b. Those plots show results for different values of n with 100 expirements per value. Red points represent the BP size and the blue line the upper bound $n + 2nH_n$.



(a) $n = 20$ (b) $n = 50$ (c) $n = 100$

(d) $n = 200$ (e) $n = 500$ (f) $n = 700$

Figure 4: Visualization of a the binary space partition size using third method of inputs generation as shown in Figure 1c. Those plots show results for different values of n with 100 expirements per value. Red points represent the BP size and the blue line the upper bound $n + 2nH_n$.

2.1 Discussion

From the results presented above, the main analysis that we can make is that the generation method of the inputs have a big impact on the binary space size. We expected the random method to not respect the upper bound due to the fact that it does not follow the definition of the problem given in the book: segments must not intersect. With these inputs, we found that for a large n, the binary space size always go beyond the upper bound.

With the generation of inputs that follows a pattern and do not intersect, the results show that the binary space size is always below the upper bound. Indeed with the two methods for input generation that we used, the upper bound is always respected. The main difference between the method with 3 segments per pattern is that the binary space size increase compare to the pattern with only 2 segments.

We conclude from these results that the inputs generation is important. The layout of the segments on the plance seems to give a higher binary space size for the same number of inputs and without going beyond the upper bound. We suppose that random generated inputs will also follow this behaviour, based on the results, if the non-intersecting segments condition is respected. Random non-intersection segements have not been implemented and it might be interesting to find a way to generate non-intersecting segments to confirm these results.

3 Source Code

3.1 Description

The binary space partition algorithm is implemented in the `bin_space` function. Generation of inputs is made in functions named `gen_<method>_inputs`. The `main` function can be configured to run multiple experiments (`nepoch`) for different n (`nvalues`) using a defined inputs generation method.

3.2 Python script

```python
#!/usr/bin/python3

import logging
import os
import sys
from datetime import datetime
from fractions import Fraction
from random import shuffle

import matplotlib.pyplot as plt
import numpy as np
from matplotlib import colors as mcolors
from matplotlib.collections import LineCollection

logging.basicConfig(level=logging.ERROR)

# Specify backend, to allow usage from terminal
plt.switch_backend('agg')


def get_slope(point0, point1):
    x1, y1 = point0
    x2, y2 = point1
    return np.divide(y2 - y1, x2 - x1)

def lin_eq(line):
    x1, y1 = line[0]
    m = get_slope(*line)
    return -m, 1, np.dot(m, x1) - y1

def intersect_point(line0, line1):
    s0, s1 = get_slope(*line0), get_slope(*line1)
    if s0 == s1:
        raise ValueError("line0: %s and line1: %s have the same slope." % (line0, line1))
    a0, b0, c0 = lin_eq(line0)
    a1, b1, c1 = lin_eq(line1)
    a = np.array(((a0, b0), (a1, b1)))
    b = np.array((-c0, -c1))
    return np.linalg.solve(a, b)

def outer_product(el, coord):
    (x1, y1), (x2, y2) = el[0], el[1]
    x, y = coord
    d = np.subtract(np.dot(x - x1, y2 - y1), np.dot(y - y1, x2 - x1))
    return d

def in_first_side(el, coord):
```

```
    return outer_product(el, coord) >= 0


def in_second_side(el, coord):
    return outer_product(el, coord) < 0

def split_space(el, elements):
    side0, side1 = [], []
    for element in elements:
        coord0, coord1 = element
        if in_first_side(el, coord0) and in_first_side(el, coord1):
            side0.append(element)
        elif in_second_side(el, coord0) and in_second_side(el, coord1):
            side1.append(element)
        elif in_first_side(el, coord0):
            intersect = intersect_point(el, element)
            side0.append((coord0, intersect))
            side1.append((intersect, coord1))
        else:
            intersect = intersect_point(el, element)
            side0.append((coord1, intersect))
            side1.append((intersect, coord0))
    return side0, side1

def _bin_space(elements):
    logging.debug('_bin_space on %s', elements)
    if len(elements) < 2:
        return len(elements)
    else:
        el = elements[0]
        elements = elements[1:]
        side0, side1 = split_space(el, elements)
        logging.debug('Split sides: %s --- %s', side0, side1)
        return _bin_space(side0) + _bin_space(side1)

def bin_space(elements):
    shuffle(elements)
    return _bin_space(elements)

def gen_random_inputs(n, seed=None):
    inputs = []
    if seed is not None:
        np.random.seed(seed)
    for i in range(n):
        coord0, coord1 = np.random.rand(2), np.random.rand(2)
        inp = (coord0, coord1)
        inputs.append(inp)
    return inputs

def gen_triangle_inputs(n):
    inputs = []
    coord0, coord1 = np.array([0.1, 0.8]), np.array([0.15, 0.95])
    big_lag, small_lag = 0.09, 0.15
    lag = 0.1
    for i in range(n):
        previous_coord0, previous_coord1 = coord0, coord1
        lag0, lag1 = (big_lag + lag, small_lag + lag) if i % 2 == 0 else (small_lag + lag, big_lag + lag)
        coord0, coord1 = np.array([coord1[0] + lag0, coord0[1]]), np.array([coord1[0] + lag1, coord1[1]])
        inp = (coord0, coord1)
        inputs.append(inp)
```

```
        new_line = coord1[0] > 10
        if new_line:
            coord0[1] -= 0.15
            coord1[1] -= 0.15
            coord0[0] = 0.15
            coord1[0] = 0.1
        else:
            inputs.append((previous_coord0, coord0))
    return inputs

def gen_twoside_inputs(n):
    inputs = []
    coord0, coord1 = np.array([0.1, 0.8]), np.array([0.15, 0.95])
    big_lag, small_lag = 0.09, 0.15
    lag = 0.1
    new_line = False
    for i in range(n):
        previous_coord0, previous_coord1 = coord0, coord1
        lag0, lag1 = (big_lag + lag, small_lag + lag) if i % 2 == 0 else (small_lag + lag, big_lag + lag)
        coord0, coord1 = np.array([coord1[0] + lag0, coord0[1]]), np.array([coord1[0] + lag1, coord1[1]])
        inp = (coord0, coord1)
        inputs.append(inp)
        new_line = coord1[0] > 10
        if new_line:
            coord0[1] -= 0.15
            coord1[1] -= 0.15
            coord0[0] = 0.15
            coord1[0] = 0.1
    return inputs

def show_inputs(inputs, results_dir, method_name):
    colors = [mcolors.to_rgba(c)
              for c in plt.rcParams['axes.prop_cycle'].by_key()['color']]
    line_segments = LineCollection(inputs, linewidths=(0.5, 1, 1.5, 2),
                                   colors=colors, linestyle='solid')
    fig, ax = plt.subplots()
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    ax.add_collection(line_segments)
    ax.set_title('Inputs example of size %s using method %s' % (len(inputs), method_name))
    fig.savefig(os.path.join(results_dir, 'inputs_sample_%s.png' % len(inputs)))

def show_result(n, results, nepoch, results_dir):
    fig = plt.figure()
    plt.plot(np.arange(nepoch), results, 'ro')
    plt.plot(np.arange(nepoch), np.full(nepoch, expected_size(n)))
    plt.title('Results for n = %s' % n)
    plt.xlabel('Runs (different inputs)')
    plt.ylabel('Size of random Binary Space Partition')
    plt.legend(['Size of BP: max %s' % max(results), r'$n + 2nH_n$'])
    f = os.path.join(results_dir, 'inputs_%s' % n)
    if os.path.exists(f + '.png'):
        i = 0
        init_f = f
        f = init_f + '_' + str(i)
        while os.path.exists(f + '.png'):
            i += 1
            f = init_f + '_' + str(i)
    fig.savefig(f + '.png')
```

```python

def expected_size(n):
    return np.add(n, 2 * np.dot(n, H(n)))

def H(n):
    return sum(Fraction(1, d) for d in range(1, n + 1))

def main():
    results_dir = os.path.join('results', str(datetime.now()))
    os.makedirs(results_dir, exist_ok=True)
    nepoch = 100
    nvalues = [2, 10, 20, 50, 100, 200, 500]
    sizes = {}
    inputs_func = gen_random_inputs
    print('Input method: %s' % inputs_func.__name__)
    for n in nvalues:
        sizes[n] = []
        print('Running for n=%s' % n)
        inputs = inputs_func(n)
        show_inputs(inputs, results_dir, inputs_func.__name__)
        for epoch in range(nepoch):
            print(epoch, end=', ')
            sys.stdout.flush()
            inputs = inputs_func(n)
            sizes[n].append(bin_space(inputs))
        print()
        print(sizes)
        show_result(n, sizes[n], nepoch, results_dir)

if __name__ == "__main__":
    main()
```

References

Outer product https://math.stackexchange.com/questions/274712/calculate-on-which-side-of-a-straight-line-is-a-given-point-located

Intersection http://infohost.nmt.edu/tcc/help/lang/python/examples/homcoord/Line-intersect.html

Harmonic Series heuristic https://stackoverflow.com/questions/404346/python-program-to-calculate-harmonic-series#404425