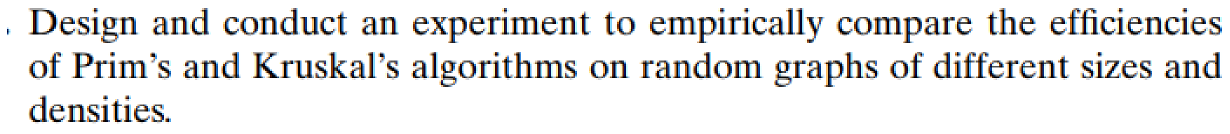
GREEDY APPROACH

GROUP 7

Nguyễn Quốc Trường 21521604

Trần Xuân Minh 21520352



, Design and conduct an experiment to empirically compare the efficiencies of Prim's and Kruskal's algorithms on random graphs of different sizes and densities.

Answer:

- Experiment Design:

The goal of this experiment is to compare the runtimes of Prim's and Kruskal's algorithms for finding minimum spanning trees on random graphs of different sizes and densities. To conduct this experiment, we will generate random graphs of various sizes and densities, and then we will run Prim's and Kruskal's algorithms on each graph and measure their runtimes. The experiment will be conducted on a computer with a standard CPU and memory configuration.

1. Graph generation:

We will generate random graphs using the Erdős–Rényi model, which generates a random graph with n vertices and a probability p of an edge between any two vertices. We will generate graphs with sizes from 10 to 100 vertices with increments of 10 vertices, and with densities from 0.1 to 0.9 with increments of 0.1.

2. Algorithm implementation:

We will implement Prim's and Kruskal's algorithms in a programming language, such as Python or Java. We will use the same implementation for both algorithms.

3. Runtime measurement:

We will measure the runtime of each algorithm on each graph using the time function in the programming language. We will run each algorithm five times on each graph and take the average runtime.

4. Analysis and comparison:

We will compare the runtimes of Prim's and Kruskal's algorithms on each graph with different sizes and densities. We will plot the results in a graph with the graph size and density on the x-axis and the runtime on the y-axis. We will also calculate the average runtime for each algorithm across all graphs with the same size or density, and we will compare the two algorithms' runtimes.

- Experimental Procedure:

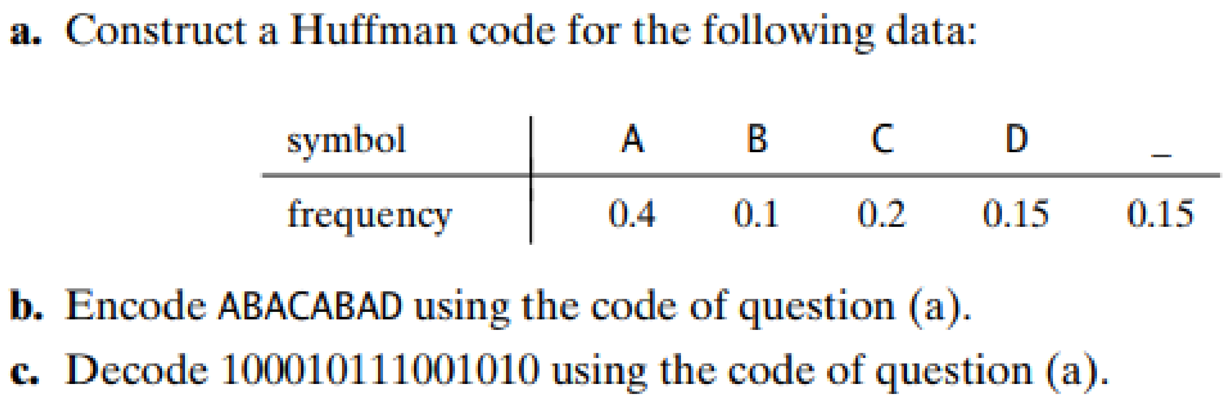
1. Generate a random graph with a given size and density using the Erdős–Rényi model.
2. Run Prim's algorithm on the graph and measure its runtime.
3. Run Kruskal's algorithm on the graph and measure its runtime.
4. Repeat steps 1-3 five times for each graph and take the average runtime of each algorithm.
5. Record the runtimes and graph sizes and densities.
6. Repeat steps 1-5 for graphs of different sizes and densities.
7. Plot the results in a graph with the graph size and density on the x-axis and the runtime on the y-axis.
8. Calculate the average runtime for each algorithm across all graphs with the same size or density.
9. Compare the two algorithms' runtimes for different graph sizes and densities.

- Expected Results:

We expect that Prim's algorithm will be faster than Kruskal's algorithm for sparse graphs (low density) because it has a lower time complexity of O(E log V) compared to Kruskal's algorithm's time complexity of O(E log E). For dense graphs (high density), we expect that Kruskal's algorithm will be faster because it has a lower constant factor in its time complexity. We also expect that the runtime of both algorithms will increase as the size of the graph increases.

- Conclusion:

This experiment will provide empirical evidence on the efficiencies of Prim's and Kruskal's algorithms for finding minimum spanning trees on random graphs of varying sizes and densities. The results will help us understand the performance characteristics of the two algorithms and when to use them for different graph types.



a. Construct a Huffman code for the following data:

symbol	A	B	C	D	_
frequency	0.4	0.1	0.2	0.15	0.15

b. Encode ABACABAD using the code of question (a).

c. Decode 100010111001010 using the code of question (a).

Answer:

a. To construct a Huffman code, we start by arranging the symbols and their frequencies in ascending order. Then, we combine the two symbols with the lowest frequencies into a single node, with a frequency equal to the sum of their frequencies. We repeat this process until we have a single tree.

Here are the steps to construct a Huffman code for the given data:

1. Arrange the symbols and their frequencies in ascending order:

symbol	frequency
B	0.1
_	0.15
D	0.15

C	0.2
A	0.4

2. Combine the two symbols with the lowest frequencies into a single node, with a frequency equal to the sum of their frequencies:

symbol	frequency
B	0.1
_	0.15
D	0.15
BC	0.3
A	0.4

3. Repeat step 2 until we have a single tree:

symbol	frequency
BCD	0.6
A	0.4

4. Assign 0 to the left branch and 1 to the right branch of each node, starting from the root. The resulting Huffman code is:

symbol	frequency	Huffman code
B	0.1	000
_	0.15	001
D	0.15	010
C	0.2	011
A	0.4	1

b. Using the Huffman code from part (a), we can encode ABACABAD as follows:

A: 1

B: 000

A: 1

C: 011

A: 1

B: 000

A: 1

D: 010

The encoded message is: 100011101100010.

c. Using the Huffman code from part (a), we can decode 100010111001010 as follows:

Start from the root of the Huffman tree.

Read the first bit, which is 1. Follow the right branch to node A.

Read the next bit, which is 0. Follow the left branch to node B.

Read the next bit, which is 0. Follow the left branch to node B.

Read the next bit, which is 0. Follow the left branch to node B. B is a leaf node, so output symbol B and return to the root.

Read the next bit, which is 1. Follow the right branch to node A.

Read the next bit, which is 1. Follow the right branch to node C.

Read the next bit, which is 1. Follow the right branch to node A. A is a leaf node, so output symbol A and return to the root.

Read the next bit, which is 0. Follow the left branch to node B.

Read the next bit, which is 1. Follow the right branch to node D. D is a leaf node, so output symbol D and return to the root.

Read the next bit, which is 0. Follow the left branch to node B. B is a leaf node, so output symbol B and return to the root.

The decoded message is: BABD.