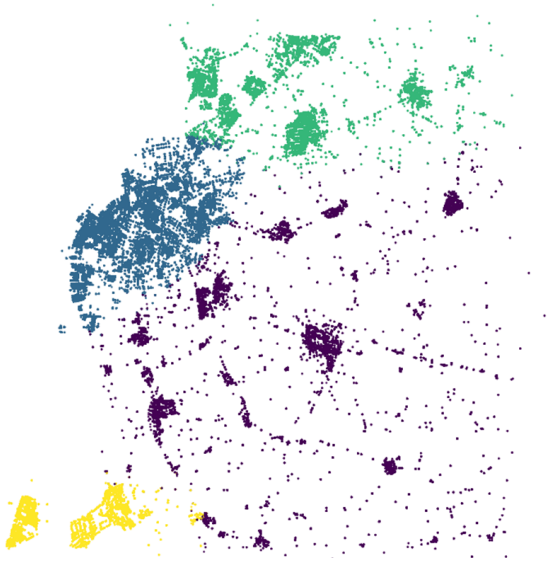
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Distributed machine learning framework for shortest path problems with stochastic weights

Master's thesis in Computer science and engineering

Gabriel Aspegrén and Olle Nilsson Dahlberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Distributed machine learning framework for shortest path problems with stochastic weights

Gabriel Aspegrén and Olle Nilsson Dahlberg

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Distributed machine learning framework for shortest path problems with stochastic weights

Gabriel Aspegrén and Olle Nilsson Dahlberg

Supervisor: Niklas Åkerblom, Department of Computer Science and Engineering
Advisor: Niklas Åkerblom, Volvo Cars
Examiner: Morteza Haghir Chehreghani, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: To the left is a map of Malmö (from Google Maps), and to the right are the intersections of Malmö, clustered into four clusters with Gaussian Mixture Model.

Typeset in LaTeX
Gothenburg, Sweden 2023

Distributed machine learning framework for shortest path problems with stochastic weights

Gabriel Aspegrén and Olle Nilsson Dahlberg
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Range anxiety is one of the most common reasons why customers hesitate to buy an electric car. At the same time, the European Union strives toward a fully electric car fleet. Previous work has shown promising results in designing a self-learning shortest path algorithm for finding the most energy efficient path for an electric vehicle through a road network, using combinatorial multi-armed bandit methods. However, it is desirable to scale the methods for larger networks, for example countries and continents. In this project, we design a distributed framework for shortest path computations on a road network, over a computer cluster, using combinatorial multi-armed bandit methods and machine learning. The system is distributed with Apache Spark and GraphX's version of the Pregel algorithm. An experimental study is performed to investigate the impact of partition strategy, number of partitions, network size and latency between computer nodes on the total run-time. The results show that partitioning strategy has an significant impact on the run-time and that larger networks benefit more from being partitioned.

Keywords: Shortest path problems, distributed computing, multi-armed bandits.

Acknowledgements

We would like to thank our supervisor Niklas Åkerblom for all support in solving different problems and challenges along the way, and for your thorough feedback on the report. We also would like to thank our examiner Morteza Haghir Chehreghani.

Finally, we would like to thank Volvo Cars and especially Ole-Fredrik Dunderberg, for the opportunity to write our master thesis at Volvo Cars, in the 96110 team.

Gabriel Aspegrén and Olle Nilsson Dahlberg, Gothenburg, 2023-06-09

Contents

List of Figures

List of Algorithms

1

Introduction

The European Union has adopted a target of reducing the carbon dioxide emission from cars by 55 % until 2030 compared to the levels of 1990. One part of this target is that, by 2035, all new cars registered in the union will be zero-emission [1]. Battery electric vehicles (BEVs) accounted for 9 % of total sales of new cars in Europe in 2021 [2]. One of the main reasons people may be disinterested in an electric vehicle is range concerns, commonly known as "range anxiety" [3]. Previous work [4], [5] has shown that a navigation system, by influencing the driver, is able to increase traffic efficiency and reduce carbon dioxide emissions from vehicles. This makes it interesting to see if it is possible to decrease energy consumption by introducing machine learning methods that take the current traffic situation into account. There have been attempts to design a self-learning navigation system for BEVs that minimizes energy consumption [6] with promising results. This was simulated on road networks of cities in Europe.

For such a navigation system to be useful, it needs to be extended to larger road networks. The method used in [6] to find the optimal path is a combinatorial multi-armed bandit algorithm combined with a shortest path algorithm. Basically, it decides which paths through the road network to explore, by utilizing a prior distribution over the expected energy consumption. The prior distributions are then updated with the observed results, in order to learn the energy consumption for the road network efficiently.

The energy consumption can be seen as stochastic due to the fact that traffic is constantly changing. The optimal path may depend on which time of the day it is, since the best path during lunch may be crowded during rush hour. To prevent sub-optimal paths from being chosen by the shortest path algorithm, continuous exploration is needed. Further, the algorithm needs to be able to handle large road networks, like countries and even continents. This requires more computational power and to solve this, one approach may be to divide the road network into partitions and distribute these to different compute nodes. If this extension of the model is effective and can extend the travel distance for electric cars it may increase the interest for BEVs.

1.1 Outline of the report

The report is structured as follows. First, we present related work to our project to give the current state of the field. Then, we formulate our research question with specified aims for the project. In Chapter 2, we present all theoretical background that is needed to understand our method and applications, with text and pseudo-code. Later, we describe our method, with focus on how we implement and combine the algorithms presented in Chapter 2. Finally, Chapters 4 and 5 provide the results and our conclusions, and also what has to be investigated in the future.

1.2 Related work

In [5], it was shown that it is possible to reduce carbon dioxide emissions from vehicles by 15 - 20 % by taking the road grade into account. This method was not self-learning and did not take the current traffic situation into account. [4] showed that a navigation system can influence the driver and improve traffic efficiency. Navigation on distributed road network graphs has been proven efficient in [7] and [8]. Reducing energy consumption by self-learning navigation and combinatorial multi-armed bandit methods [6] has shown promising results. Google [9] has introduced a distributed graph processing framework called Pregel that can be used for finding shortest paths in graphs. Apache Spark is an open-source platform, suitable for large-scale data processing in parallel on computer clusters, as well as iterative machine learning tasks [10], [11]. Further, Spark offers a graph component, called GraphX [12], which allows parallel graph computations. The combinatorial multi-armed bandit problem approach to finding the most energy efficient path, in a distributed road network, run with Apache Spark on a computer cluster has, to our knowledge, not been studied in any prior work.

1.3 Aims

The goal of this project is to develop methods that can reliably, using observations acquired over time, estimate the shortest path of a large network with stochastic weights with reasonable run-time.

The aim of this project is to answer the following question: *Is it possible to, by partitioning a road network and distributing the computations over a computer cluster, find the most energy efficient path using combinatorial multi-armed bandit methods and machine learning while reducing the total run-time?*

To answer this, the following components are developed:

- A shortest path algorithm, which is able to accurately calculate the least energy consuming path of a network.
- An online machine learning algorithm capable of utilizing the observations

made with the shortest path algorithm to estimate the underlying energy consumption distributions.

- A distributed system, which the shortest path and online machine learning algorithms can be integrated into, which improves run-time.

The performance of the methods are evaluated in an experimental study where total run-time is used as metric.

2

Theory

In this section, all relevant background for the methods and techniques used in the project is explained.

2.1 Graphs

A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a set of ordered pairs such that $\mathcal{E}$ are 2-element subsets of $\mathcal{V}$ [13]. The elements of $\mathcal{V}$ are called *vertices* and the elements of $\mathcal{E}$ are called *edges*, where the number of edges connected to each vertex $v \in \mathcal{V}$ is called the *degree* of v. In this project, directed graphs are studied, which means that if the vertex $v_1 \in \mathcal{V}$ is connected to $v_2 \in \mathcal{V}$ with an edge, it does not necessarily mean that $v_2 \in \mathcal{V}$ is connected to $v_1 \in \mathcal{V}$. A graph is a good way to structure data if there are underlying relationships, e.g., the relations in a social network. Both vertices and edges can have associated attributes. These attributes vary depending on what information is important to the analysis. For example, if the graph is a road network, the vertex attributes could be coordinates, while an edge attribute may be the time required to traverse the edge.

2.2 Shortest path algorithms

When examining the paths of a network it is essential to have a way to quickly find the optimal path from one vertex to another. The types of algorithms which are designed to optimize this problem are called shortest path algorithms, since the edge attributes typically represent the distance between vertices. There are multiple algorithms with different advantages and disadvantages, and choosing the right one can have a large effect on the performance and run-time of the method.

A well-known shortest path algorithm is Dijkstra's algorithm [14] which for a given source vertex finds the shortest paths to all other vertices in the network. First, a list of all the end vertices of the paths currently explored and their distance to the source vertex is initialized. The neighbours of the end vertex with the current shortest distance to the source vertex are then explored and new end vertices are added to the list. This continues until the target vertex is found, and the distance back to the source vertex is obtained. The algorithm can be terminated when the

shortest path to the target vertex has been determined and therefore prevent unnecessary computations from being performed. Dijkstra's algorithm is summarized in Algorithm 1.

Algorithm 1 Dijkstra's algorithm

procedure DIJKSTRA(Graph $\mathcal{G}$, Source $v_s \in \mathcal{V}$)
  List of distances to source vertex Dist
  List of predecessor vertices Prev
  Set of potential shortest path vertices $\mathcal{V}_{\text{potential}} \leftarrow \emptyset$
  for each vertex $v \in \mathcal{V}$ **do**
    $\text{Dist}[v] \leftarrow \infty$
    $\text{Prev}[v] \leftarrow$ Nothing
  end for
  $\mathcal{V}_{\text{potential}} \leftarrow \mathcal{V}_{\text{potential}} \cup \{v_s\}$
  $\text{Dist}[v_s] \leftarrow 0$
  while $\mathcal{V}_{\text{potential}}$ is not empty **do**
    $v_c \leftarrow \arg\min_{v \in \mathcal{V}_{\text{potential}}} \text{Dist}[v]$
    if $v_c = v_d$ **then**
      break
    end if
    $\mathcal{V}_{\text{potential}} \leftarrow \mathcal{V}_{\text{potential}} \setminus v_c$
    for each outgoing neighbor v_n of v_c **do**
      $w_{c,n} \leftarrow$ Weight of edge from v_c to v_n
      $l \leftarrow \text{Dist}[v_c] + w_{c,n}$
      if $l < \text{Dist}[v_n]$ **then**
        $\text{Dist}[v_n] \leftarrow l$
        $\text{Prev}[v_n] \leftarrow v_c$
        if v_n is not in $\mathcal{V}_{\text{potential}}$ **then**
          $\mathcal{V}_{\text{potential}} \leftarrow \mathcal{V}_{\text{potential}} \cup \{v_n\}$
        end if
      end if
    end for
  end while
  return Dist, Prev
end procedure

Another shortest path algorithm that is widely used is the A* search algorithm [15]. A* is dependant on a heuristic function HEURISTICFUNCTION, which estimates the distance from the current vertex to the target vertex. At each vertex, an estimation of the distance to the target vertex is made, based on the traversed distance and the heuristic function. The algorithm chooses the path with the shortest estimated distance. With different heuristic functions, accuracy or speed can be favored. Unlike Dijkstra's algorithm, A* only finds the shortest path from the source vertex to the target vertex and does not store the shortest distance to all other vertices. The A* algorithm is seen in Algorithm 2.

Algorithm 2 A* algorithm

procedure A*(Graph $\mathcal{G}$, Source $v_s \in \mathcal{V}$, Destination $v_d \in \mathcal{V}$, HeuristicFunction)
  List of distances to source vertex Dist
  List of estimated distances to target vertex EstimatedTotal
  List of predecessor vertices Prev
  Set of potential shortest path vertices $\mathcal{V}_{\text{potential}} \leftarrow \emptyset$
  for each vertex $v \in \mathcal{V}$ **do**
    Dist[v] $\leftarrow \infty$
    EstimatedTotal[v] $\leftarrow \infty$
    Prev[v] $\leftarrow$ Nothing
  end for
  $\mathcal{V}_{\text{potential}} \leftarrow \mathcal{V}_{\text{potential}} \cup \{v_s\}$
  Dist[v_s] $\leftarrow 0$
  EstimatedTotal[v_s] $\leftarrow$ HeuristicFunction(v_s)
  while $\mathcal{V}_{\text{potential}}$ is not empty **do**
    $v_c \leftarrow \arg\min_{v \in \mathcal{V}_{\text{potential}}}$ EstimatedTotal[v]
    if $v_c = v_d$ **then**
      break
    end if
    $\mathcal{V}_{\text{potential}} \leftarrow \mathcal{V}_{\text{potential}} \setminus v_c$
    for each outgoing neighbor v_n of v_c **do**
      $w_{c,n} \leftarrow$ Weight of edge from v_c to v_n
      $l \leftarrow$ Dist[v_c] + $w_{c,n}$
      if $l <$ Dist[v_n] **then**:
        Dist[v_n] $\leftarrow l$
        EstimatedTotal[v_n] $\leftarrow$ Dist[v_n] + HeuristicFunction(v_n)
        Prev[v_n] $\leftarrow v_c$
        if v_n is not in $\mathcal{V}_{\text{potential}}$ **then**
          $\mathcal{V}_{\text{potential}} \leftarrow \mathcal{V}_{\text{potential}} \cup \{v_n\}$
        end if
      end if
    end for
  end while
  return Dist, Prev
end procedure

2.3 Multi-armed bandit problems and algorithms

One approach to finding the optimal path when the edge weights are stochastic and the distributions are initially unknown is to implement a multi-armed bandit algorithm. These types of algorithms are occasionally used for combinatorial problems where each possible choice has an individual, unknown probability of being the best, and where exploration is required to learn the optimal choice. This type of problem is based on a setting where there is a set $\mathcal{A}$ of multiple slot machines

(or arms) available, each machine $a \in \mathcal{A}$ having an individual, unknown probability of resulting in a win. If one has a finite amount of rounds T to play and multiple arms to choose from, a multi-armed bandit algorithm offers a strategy to find the best arm, in terms of expected payoffs. A trade-off between searching for the best arm, called *exploration*, and playing the best arm found, called *exploitation* has to be made, since the number of rounds is limited [16]. An algorithm that chooses the best feasible arm, given expected rewards is called an oracle, and is a crucial part of the multi-armed bandit algorithm. The multi-armed bandit algorithm is outlined in Algorithm 3.

Algorithm 3 Multi-armed bandit algorithm

for $t \in [1, T]$ **do**
  Select arm $a_t \in \mathcal{A}$
  Receive reward $r_t(a_t)$
  Use observed reward $r_t(a_t)$ to update knowledge
end for

A Bayesian greedy multi-armed bandit algorithm takes the expected reward (assumed to be sampled from a prior distribution) for each arm and exploits the arm with the highest value. After each round, the prior is updated with the observed reward and is used as the new prior in the next round. The greedy algorithm results in a low level of exploration. An extension of this algorithm is the ϵ-greedy method. Each round, it exploits the arm with the highest estimated expected reward with a probability of $1 - \epsilon$, and otherwise it explores a random arm, with a usually small probability ϵ. This extension of the algorithm often finds better arms but does not explore in an informative way [17].

Thompson Sampling was first introduced in 1933 by William R. Thompson [18], but not popularized until circa 2010 [17], [19]. Like the ϵ-greedy algorithm, the Thompson Sampling method is both exploring and exploiting, but the exploration is done in an informative manner. It samples expected rewards from the prior distributions and chooses the arm with the highest sampled expected reward, while observing the reward revealed by the environment. This way, the algorithm explores arms with the potential of being the optimal arm according to the prior beliefs. Previous work [20] has shown that the Thompson Sampling method is outperforming greedy algorithms in finding the optimal arm. Thompson Sampling for the stochastic shortest path problem is outlined in Algorithm 4.

Another bandit strategy is called the Bayesian Upper Confidence Bound algorithm, Bayes-UCB [21], [22]. This is an optimistic approach to the multi-armed bandit problem [23]. This algorithm is usually used to maximize reward, but since the goal is to find the shortest path, the algorithm has been modified to choose the arm with the smallest lower confidence bound, which is a probable underestimation of the expected mean. Each time an arm is chosen, the confidence bound narrows, which may result in another arm having the lowest confidence bound in the next round and may therefore be chosen next. The accuracy of the confidence bound increases as an arm is played. In this manner, an arm can only be chosen if the confidence bound is

Algorithm 4 Thompson sampling

procedure ThompsonSampling(Graph $\mathcal{G}$, Source $v_s \in \mathcal{V}$, Destination $v_d \in \mathcal{V}$, Prior parameters $\theta_{e,0}$ for $e \in \mathcal{E}$)
  for $t \in [1, T]$ **do**
    for each edge $e \in \mathcal{E}$ **do**
      Use parameters $\theta_{e,t-1}$ to sample edge weight from distribution
    end for
    Use shortest path algorithm to find shortest path between v_s and v_d with respect to sampled edge weights
    Traverse the shortest path to observe reward r_e for each e
    for each e in shortest path **do**
      Update $\theta_{e,t}$ according to r_e
    end for
  end for
end procedure

lower than for the optimal arm, and over time the bounds will concentrate, leaving the optimal arm with the smallest lower bound (in expectation). Bayes-UCB for the stochastic shortest path problem is outlined in Algorithm 5.

Algorithm 5 Bayes-UCB

procedure Bayes-UCB(Graph $\mathcal{G}$, Source $v_s \in \mathcal{V}$, Destination $v_d \in \mathcal{V}$, Prior parameters $\theta_{e,0}$ for $e \in \mathcal{E}$)
  for $t \in [1, T]$ **do**
    for each edge $e \in \mathcal{E}$ **do**
      Use parameters $\theta_{e,t-1}$ to calculate the Upper Confidence Bound edge weight
    end for
    Use shortest path algorithm to find shortest path between v_s and v_d with respect to Upper Confidence Bound edge weights
    Traverse the shortest path to observe reward r_e for each e
    for each e in shortest path **do**
      Update $\theta_{e,t}$ according to r_e
    end for
  end for
end procedure

One way of implementing a Bayesian version of UCB is to adapt the Gaussian Process Upper Confidence Bound introduced in [24], which can be used to calculate the edge weights according to Equation 2.1 with a modification to calculate the lower confidence bound

$$w_{e,t} = \mu_{e,t-1} - \sqrt{\beta_t}\sigma_{e,t-1}, \quad \beta_t = 2ln\left(\frac{(t^2+1)|\mathcal{A}|}{\sqrt{2\pi}}\right). \tag{2.1}$$

If the prior distribution is described by a Gaussian distribution, $\mu_{e,t-1}, \sigma^2_{e,t-1}$ are the prior distribution parameters for edge e and $\mathcal{A}$ is the action space.

Another method [22]of calculating the lower confidence bound is by using the quantile function with the posterior distribution, where the value for which a certain percentile of the distribution is less than, is used as the edge weight.

To evaluate the performance of the algorithms, one can evaluate the *regret*. It is the difference in expected reward between the actual optimal arm and the arm selected by the algorithm, and is defined in Equation 2.2 as

$$\text{Regret(T)} = \mathbf{E}\left[\sum_{t\in[T]} (\mathbf{E}[r_t(a^*)] - \mathbf{E}[r_t(a_t)])\right], \tag{2.2}$$

where a^* is the optimal arm and a_t is the arm proposed by the algorithm. The sum is over each round $t \in [T]$.

2.4 Distributed computing

The memory and number of computations required for the methods outlined above increases as the network increases in size, hence the computational demands eventually becomes too much when executed on a single machine. To deal with this problem, distributed computing may be utilized. One way to apply distributed computing is to partition a data set into smaller segments, enabling that each partition can be operated upon as an individual data set. A cluster of compute nodes is then used to perform computations in parallel rather than in sequence. One can think of it as multiple computers calculating different parts of the same calculation simultaneously, but since they only have access to some of the data locally, they have to request and receive information from the other compute nodes. This communication between the nodes is quite costly in terms of time delay [25], compared to operations within a single compute node. Therefore this method is used when dealing with large data sets, since the benefit from parallelization has to outweigh the added communication overhead.

To further reduce the run-time, one can examine how the partitioning is performed, since an optimized partitioning scheme can lead to fewer communications between partitions. The number of partitions and how to divide the data in relation to the size and structure of the network have to be investigated to find an optimal partition strategy.

In 2010, Google released a model for large-scale graph processing called Pregel [9]. This model has many applications and the implementation differ with each. For the shortest path problem, the computations in Pregel are performed in a sequence of iterations, called a *superstep*. In each superstep, vertices are sending and receiving messages and updating their statuses (active or inactive). A superstep includes receiving and combining messages, checking and updating vertex status,

and sending messages. When no vertices are active, the algorithm is terminated. The PREGEL computational model is outlined in Algorithm 6.

Algorithm 6 Pregel

procedure PREGEL(Graph $\mathcal{G}$)
Set initial state, value and messages of each vertex $v \in \mathcal{V}$
while active messages remain **do**
for all $v \in \mathcal{V}$ with active messages (in parallel) **do**
Receive and aggregate messages for ingoing edges from neighboring vertices to v, according to user defined transformation and aggregation functions
Update state and value of v, and send messages through outgoing edges to neighbouring vertices
end for
end while
return Transformed graph $\mathcal{G}'$
end procedure

2.4.1 Apache Spark

Apache Spark is a software framework for large-scale data processing [26]. By distributing processing work to a cluster of computing nodes, it offers the user efficient tools for distributed programming. The reason for Sparks ability to handle large data sets with ease is its architecture. Spark is a cluster computing framework and offers a partitioned collection of elements called a Resilient Distributed Data set, *RDD* [27], allowing parallel computations to be performed on the elements. New RDDs can be created through *transformations* of existing RDDs, with operations like MAP and FILTER. The MAP operation passes the source RDD through a specified function, applying it to each element in parallel before building the new RDD, based on the function value. In the same way, FILTER passes the source RDD through a filter function and creates a new RDD containing only the elements for which the condition is *true* [28]. The elements in the RDDs can also be aggregated using *actions* like REDUCE which aggregate elements of data sets with a commutative and associative binary function, which takes two arguments and returns a single value.

A Spark cluster consists of one master and multiple worker nodes. The master receives instructions from the application, and forwards the instructions to the workers while monitoring all activity [29]. Partitions can be distributed to different workers by the master. When operations require data to be shared between partitions, Spark performs a *Shuffle* operation, which consists of three steps [27], [28]. First, the MAP transformation is used to partition the data based on a key. Then the shuffle operation distributes data between nodes, grouping data with the same key. This is a costly process that requires moving a lot of data. Lastly, the REDUCE action is performed to aggregate the data.

When running an application, Spark has two ways of scheduling *jobs* [30], where a job is an action, like REDUCE, and the tasks needed to perform that action. Jobs get

divided into *stages*, e.g., map and reduce phases, and are run in the order launched. This is known as a FIFO-scheduler (First In First Out), and the first job gets all available cluster resources, while the second job gets priority next. If a job does not need all resources, the next job is launched with the unused resources. The second scheduler is called FAIR, and it assigns an equal share of resources to all jobs and allows launching jobs even if long jobs are still running.

As mentioned, RDDs are data sets divided into partitions. A graph can be partitioned in different ways and two common methods are vertex-cut and edge-cut [31] which differ in where to draw the partition border, through vertices or through edges. Spark offers a graph processing framework called *GraphX* [12] which utilizes vertex-cut [32]. Further, GraphX uses RDDs to allow parallel computing on graphs, and also enables a Pregel API. Graphs are constructed of an EdgeRDD and a VertexRDD. The edge and vertex can be combined into a triplet, which contains the attributes of the edge and its connected vertices. These triplets are stored in an additional RDD associated with the graph.

The GraphX Pregel API comes with the ability to cache RDDs to prevent the system from rebuilding RDDs, hence reducing execution time [12]. When run on a distributed system, the cached data must be available to all compute nodes. Apache has designed a distributed file system [33] which makes stored data available to all workers in the cluster.

2.4.2 Docker

Docker is a software that utilizes the concept of containers, where a container is a software environment that consists of all the code and dependencies required for running a specific process or application [34]. A machine can run multiple containers with different operating systems and applications and it is also possible to run a single application, distributed on multiple containers. Containers offer a way to isolate single or multiple specific environments on a single machine and the Docker software enables the user to setup and manage containers easily. It is possible to set up a virtual network with multiple containers as a Spark cluster, with containers assigned as master and workers on single or multiple actual machines. To run an application with Docker, an image is needed. An image is a package with everything needed to run the application, such as how to initiate the containers and which libraries to add.

2.5 Data

Data sets of road networks as graphs are available as open source through OpenStreetMap [35]. They contain the coordinates of each road segment and intersection, as well as characteristics such as length, speed limit, mean time to traverse, and mean energy consumption for each road segment, and are pre-processed to filter out unnecessary information. The mean energy consumption is calculated based on the assumptions made in [6]. The resulting graphs are of low degree, since no vertex

has more than 8 neighbors. These maps come in varying sizes, but Sweden is represented by a graph with 680 000 vertices and 1 600 000 edges, as a reference. These characteristics are important to recognize since they can help identify the optimal partitioning strategy [36].

2.5.1 Partitioning

A rule of thumb in partitioning is to have equally sized partitions [37]. A graph of a road network may be heterogeneous due to the number of connections inside a city being much greater than between cities. Furthermore, it is desirable to have as few connections between partitions as possible since these may be between computer nodes in a cluster and will increase communication time. These two conditions make a heterogeneous graph harder to partition. To reduce communication between partitions one may want to have cities in different partitions, since connections between cities most likely are fewer than within, but this will probably result in unequally sized partitions.

3

Methods

The programming language used in this project is Scala, since the Spark framework is constructed using it. The distributed combinatorial multi-armed bandit algorithm can be generalized as follows: The data is pre-processed, converted into a graph and partitioned. Then, in each iteration, weights are generated from the prior distributions using a specified bandit strategy to create a static network which can be used as input in the shortest path algorithm. The observed rewards from traversing the path are then evaluated and the prior distributions are updated accordingly. This is run locally on a Spark cluster.

3.1 Data pre-processing

A CSV file containing data of the Swedish road network is read and converted into a GraphX graph with energy consumption as edge attributes (derived using Equation 2.1 in [6]). Geographical restriction is applied to reduce the network to a manageable size, due to limited computing resources.

Like in [6], the energy consumed by traversing a road segment is assumed to be Gaussian distributed, with the mean μ_e and the variance $\sigma^2_{e,noise}$, due to its dependency on multiple statistically independent factors. We also assume that $\sigma^2_{e,noise}$ is fixed and known, and that the energy consumption of different edges is mutually independent. Also as performed in [6], we take a Bayesian approach and assume that μ_e is Gaussian distributed with prior parameters $\mu_{e,0}$ and $\sigma^2_{e,0}$, determined by the mean consumption from the data set. In line with this assumption, we sample μ_e for all $e \in \mathcal{E}$ from the prior distributions at the beginning of each experiment to act as the underlying reward distributions of the environment.

3.2 Partitioning

GraphX in Spark offers four different partitioning strategies; RandomVertexCut (RVC), CanonicalRandomVertexCut (CRVC), EdgePartition1D (EP1D) and EdgePartition2D (EP2D). RVC and CRVC partitions the edges arbitrarily across partitions by hashing the source and vertex ID of the edge, where CRVC place edges between the same vertices into the same partition, regardless of direction, unlike RVC. EP1D and EP2D partitions the edges based on the IDs of its vertices.

While EP1D only considers the source vertex of the edge, EP2D considers both the source and the target vertex. These four strategies are compared against each other and the best one is used in the intricate tests and represents a vertex ID-based partition strategy.

One hypothesis is that cross-partition communications can be reduced if the edges are partitioned based on their geographical location. EP1D and EP2D partitions are based on source and target vertex IDs, but the vertex IDs have no geographical relation. Since the coordinates of each vertex are available, it should be possible to utilize them to create a better partition strategy.

One approach to partition the data is to use clustering. If the data is clustered based on their geographical position, edges close to each other, i.e. in cities, should be partitioned together to a higher extent than by random partitioning. The KMEANS method aims to find cluster centers that minimize the sum of squared distances from each data point in the cluster to the center point [38]. Another clustering method is Gaussian Mixture Model clustering (GMM) [39] which is a probabilistic method. The data is assumed to be generated from different Gaussian distributions and the model is trying to estimate the parameters of each distribution. The data points are clustered to the distribution that has the highest probability of generating the points.

3.3 Multi-armed bandit framework

The first step of our distributed Combinatorial Multi-Armed Bandit (CMAB) Algorithm, seen in Algorithm 7, is the initialization of a distributed GraphX graph, described in 3.1, with the prior distribution parameters as edge attributes. Then, for each time step $t \in [1, T]$ a set of parallel Spark operations follows, which distributes the combinatorial bandit algorithm from [6] across partitions. The operations are selected in such a way that no communication is required between these partitions, except for the shortest path computation, where PREGEL is utilized.

By applying the map function BANDITSTRATEGY onto all edges of the graph, each edge is assigned an additional attribute, which is used as the edge weight for the shortest path computation.

BANDITSTRATEGY uses either THOMPSON SAMPLING or BAYES-UCB to generate this attribute according to Equation 2.1.

This graph is then, together with the source vertex $v_{s,t}$, used in Algorithm 9 to find the shortest path from $v_{s,t}$ to all other vertices. The graph PathGraph is returned, in which the attributes of each vertex contain a list with all the vertices in the shortest path to the source vertex and the total sum of that path's edge weights. Since only the path between $v_{s,t}$ and the destination vertex $v_{d,t}$ is of interest, FILTER is applied on PathGraph to return a VertexRDD with only one element, called PathRDD.

The Spark CARTESIAN operation returns the cartesian product of the single-element PathRDD and the VertexRDD of the original graph PriorGraph, to create PathVer-

texRDD, where each vertex has information regarding the shortest path from $v_{s,t}$ to $v_{d,t}$. PathVertexRDD is then combined with the EdgeRDD of PriorGraph to create OracleGraph. The UpdatePosterior function is then applied onto all elements in the TripletRDD of OracleGraph in parallel, using the Map operation.

UpdatePosterior examines if the edge e is part of the shortest path from $v_{s,t}$ to $v_{d,t}$. In that case, a reward r_e is observed from the environment, which is used to update the posterior distribution parameters of e.

Algorithm 7 Distributed Combinatorial Multi-Armed Bandit Algorithm

procedure DISTRIBUTED-CMAB-ALGORITHM
PriorGraph ← Initialization of GraphX graph (VertexRDD and EdgeRDD), with no vertex attributes and with edge attributes for each $e \in \mathcal{E}$ set to the prior parameters $\mu_{e,0}, \sigma^2_{e,0}$
for $t \in [1, T]$ **do**
BanditGraph ← PriorGraph.MAPEDGES(BANDITSTRATEGY)
Receive current source $v_{s,t}$ and destination $v_{d,t}$ vertices from environment
PathGraph ← SHORTESTPATHGRAPH(BanditGraph, $v_{s,t}$)
PathRDD ← PathGraph.VertexRDD.FILTER($v = v_{d,t}$)
PathVertexRDD ← PriorGraph.VertexRDD.CARTESIAN(PathRDD)
OracleGraph ← Graph(PathVertexRDD, PriorGraph.EdgeRDD)
PosteriorGraph ← OracleGraph.TripletRDD.MAP(UPDATEPOSTERIOR)
PriorGraph ← PosteriorGraph
end for
return Graph
end procedure

procedure BANDITSTRATEGY(e)
$(\mu_{e,t-1}, \sigma^2_{e,t-1}) \leftarrow$ Attributes of e
if using Thompson Sampling **then**
Use parameters $\mu_{e,t-1}, \sigma^2_{e,t-1}$ to sample edge weight w_e from posterior distribution
else if using Bayes-UCB **then**
Use parameters $\mu_{e,t-1}, \sigma^2_{e,t-1}$ to calculate the Upper Confidence Bound edge weight w_e
end if
return New attributes of e: $(\mu_{e,t-1}, \sigma^2_{e,t-1}, w_e)$
end procedure

procedure UPDATEPOSTERIOR((v_1, e, v_2))
$p_{v_1} \leftarrow$ Attribute of v_1
$(\mu_{e,t-1}, \sigma^2_{e,t-1}) \leftarrow$ Attributes of e
if v_1 and v_2 is part of p_{v_1} **then**
Observe reward r_e from environment
$\sigma^2_{e,t} = \frac{1}{\frac{1}{\sigma^2_{e,t-1}} + \frac{1}{\sigma^2_{e,noise}}}$
$\mu_{e,t} = \sigma^2_{e,t} * (\frac{\mu_{e,t-1}}{\sigma^2_{e,t-1}} + \frac{r_e}{\sigma^2_{e,noise}})$
else
$\sigma^2_{e,t} \leftarrow \sigma^2_{e,t-1}$
$\mu_{e,t} \leftarrow \mu_{e,t-1}$
end if
return New attributes of e: $(\mu_{e,t}, \sigma^2_{e,t})$
end procedure

3.4 Shortest path algorithm as oracle

Since the edges of the network describe energy consumption, it is possible for the edge values to be negative i.e., traversing a road segment generates energy. However, since the energy consumption is stochastic, this may results in negative loops, hence traversing that path would generate energy endlessly, breaking the law of conservation of energy. Therefore, we restrict the edge weights to be positive.

A heuristic function is hard to design, since the energy consumption is a combination of many factors, some being stochastic. This, together with the fact that it is necessary that the heuristic function never overestimates the consumption, makes the A* search algorithm difficult to implement.

The oracle is therefore based on a distributed version of Dijkstra's algorithm, utilizing the GraphX version [12] of the Pregel algorithm, which can be seen in Algorithm 8.

Algorithm 8 Apache Spark GraphX Pregel

```
procedure Pregel(Graph, InitialMessage, VProg, SendMsg, MergeMsg)
    Graph ← Graph.MapVertices(VProg(InitialMessage))
    MessageRDD ← Graph.TripletRDD.Map(SendMsg)
    MessageRDD ← MessageRDD.Reduce(MergeMsg)
    while MessageRDD not empty do
        Graph ← Graph.JoinVertices(MessageRDD, VProg)
        MessageRDD ← Graph.TripletRDD.Map(SendMsg)
        MessageRDD ← MessageRDD.Reduce(MergeMsg)
    end while
    return Graph
end procedure
```

In the first step of the Pregel algorithm, an initial message is sent to each vertex. MapVertices applies the function VProg, with respect to the initial message, onto the attributes of every vertex of the graph. SendMsg is then applied onto all elements in the TripletRDD of the updated graph to create messages, which are stored in MessageRDD. The reduce function MergeMsg is applied onto MessageRDD, where if a vertex receives multiple messages, MergeMsg reduces those to one.

A loop then starts, which continues as long as MessageRDD is not empty. JoinVertices joins the vertices of the graph with the messages in MessageRDD and is followed by the map function VProg onto the joined vertices. This results in a new VertexRDD which replaces the previous one in the graph. Map and Reduce is then performed the same way as previously (with SendMsg and MergeMsg, respectively) to create a new MessageRDD.

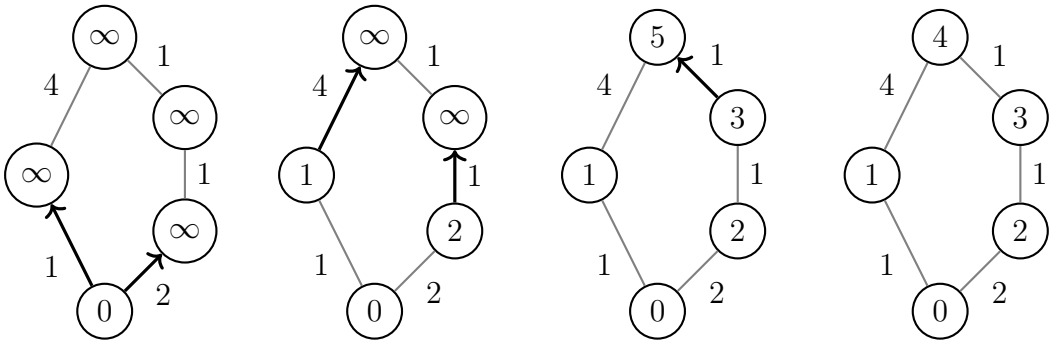
The usage of Pregel in a single source shortest path algorithm (based on the algortihm outlined in [12]) can be seen in Algorithm 9 . With InitializeGraph, each vertex $v \in \mathcal{V}$ in the network is assigned two attributes. The first attribute

Algorithm 9 Shortest Path Graph

```
procedure ShortestPathGraph(Graph, source vertex v_s)
    Graph ← Graph.MapVertices(InitializeGraph)
    InitialMessage ← (∞, List[ ])
    return Pregel(Graph, InitialMessage, vprog, sendMsg, mergeMsg)
end procedure

procedure InitializeGraph(v)
    p_v ← List[ ]
    if v = v_s then
        d_v ← 0
    else
        d_v ← ∞
    end if
    return New attributes of v: (d_v, p_v)
end procedure

procedure VProg(v, Message)
    (d_v, p_v) ← Attributes of v
    (d_Message, p_Message) ← Attributes of Message
    if d_Message < d_v then
        return Attributes of Message
    else
        return Attributes of v
    end if
end procedure

procedure MergeMsg(Message_1, Message_2)
    (d_Message1, p_Message1) ← Attributes of Message_1
    (d_Message2, p_Message2) ← Attributes of Message_2
    if d_Message1 < d_Message2 then
        return Attributes of Message_1
    else
        return Attributes of Message_2
    end if
end procedure

procedure SendMsg((v_1, e, v_2))
    (d_v1, p_v1) ← Attributes of v_1
    (d_v2, p_v2) ← Attributes of v_2
    w_e ← Attribute of e
    if d_v1 + w_e < d_v2 then
        d_Message ← d_v1 + w_e
        p_Message ← Append v_1 to p_v1
        return Attributes of Message: (d_Message, p_Message)
    else
        return No Message
    end if
end procedure
```

p_v is a list which stores the vertices that are part of the shortest path from v_s to v, making backtracking possible. The second attribute, d_v is the accumulated edge weights of the current shortest path from the source vertex. Each edge $e \in \mathcal{E}$ has one attribute, w_e which denotes the length of the edge. After the graph is initialized, the initial message is created, which is sent to all vertices in the beginning of the algorithm.

The function VProg replaces the attributes of a vertex if an incoming message contains a shorter path than its current shortest path. If a vertex receives multiple messages, MergeMsg checks which of them contains the shorter path and passes it on to the vertex. SendMsg sends messages to neighboring outgoing vertices if a shorter path can be obtained. This process is visualized in Figure 3.1.



(a) Iteration 1 **(b)** Iteration 2 **(c)** Iteration 3 **(d)** Iteration 4

Figure 3.1: A schematic overview of the Pregel shortest path algorithm in a network with five vertices over three iterations. The edge numbers represent their weights and the vertex number is the shortest path to the source vertex. Black arrows indicates messages being sent.

3.5 Experimental setting

There are many different parameters to vary in the experimental setting. The approach is first to establish which multi-armed bandit strategy performs best and then test different configurations, adjusting network size, number of partitions and partitioning strategy. The model is run locally, therefore different levels of latency are added to simulate a network environment. The setup for all experiments is found in Appendix A.

3.5.1 Choosing multi-armed bandit strategy

To measure the performance of Thompson Sampling and Upper Confidence Bound, cumulative regret is calculated according to Equation 2.2. The best-performing algorithm is then used for the remaining experiments. The same network with the same source and target vertices is used throughout the tests. Using the same vertices results in fewer road segments to explore and fewer parameters to adjust and

therefore reduces the time steps needed to observe if an algorithm exhibits sub-linear regret.

3.5.2 Testing network and cluster parameters

There are many metrics which can be used to measure the performance of the algorithm. Time is a good metric since it is expected to be related to the number of cross-partition communications. The number of cross-partition communications is affected by the size of the graph, the partitioning strategy and the number of partitions, and the cross-partition communication time is increased as more latency is added. The experiments are therefore designed to vary graph size, partition strategy, number of partitions and latency between partitions, and see how the run-time is affected. The run-time differs somewhat between experiments due to fluctuations in available resources, hence an average of six runs is used. Only one parameter is changed at a time in the experiments.

The Sweden road network consists of approximately 1.6 million edges. Due to limited computing resources, the network is geographically restricted, and experiments are performed on sizes of 5 000, 40 000, and 100 000 edges. Some additional experiments with other regions are carried out as well.

3.5.3 The Spark cluster

A Spark cluster is set up locally in a Docker environment with a total of 20 GB of memory and 16 cores, with one worker node per container. The container cluster uses a pre-built Apache Spark Docker image [40]. The number of workers is equal to the number of partitions in all experiments. The master memory is set to 4 GB, while the remaining 16 GB is evenly distributed across the worker nodes, which means that the total memory is equal for all experiments. To simulate a network environment, some latency is added to all containers.

4

Results

The aim of the project is to create a framework able to learn and find the least energy-consuming path, while also doing so in a distributed way and with reasonable run-time. Each aspect is individually evaluated and the results are presented as graphs.

4.1 Learning the least energy-consuming path

To evaluate the framework's performance using Thompson Sampling and Bayes-UCB as the bandit strategy, only a fraction of the Swedish road network was used. Since many iterations are needed to evaluate the performance of the bandit strategies, a smaller network was used to get keep the run-time manageable. Since the framework's performance is highly dependent on the relation between the prior distributions and the underlying reward distributions, multiple levels of variances were compared.

The prior and noise variances are assumed to be proportional to the approximated energy consumption $\mu_{e,0}$ and are scaled with a factor such that $\sigma^2_{e,0} = (\phi\mu_{e,0})^2$ and $\sigma^2_{e,noise} = (\phi\mu_{e,0})^2$. Throughout each test, the scaling factor for the prior and noise variances remained the same and the different values used were $\phi = (0.05, 0.1, 0.2)$.
In Figure 4.1 UCB has a much higher cumulative regret for all levels of variance. For all levels of variance, for both UCB and TS, there is a large increase in cumulative regret the first 100-200 time steps and then the slope decreases and the regret saturates at different rates, generally quicker for lower variances.

4.2 Run-time of the framework

The results from the experiments testing out partitioning strategy, network size and latency for different numbers of partitions while measuring the run-time are shown in Figure 4.3. All data points are an average of 6 repetitions. One experiment was run with 1 partition, with one worker with zero latency to the master node and 40 000 edges, and the run-time average was 49.7 seconds.

The GraphX built-in partitioning strategies were run on a network of size 40 000 edges with 4 partitions and 4 workers each with 4 GB memory. The driver was allo-

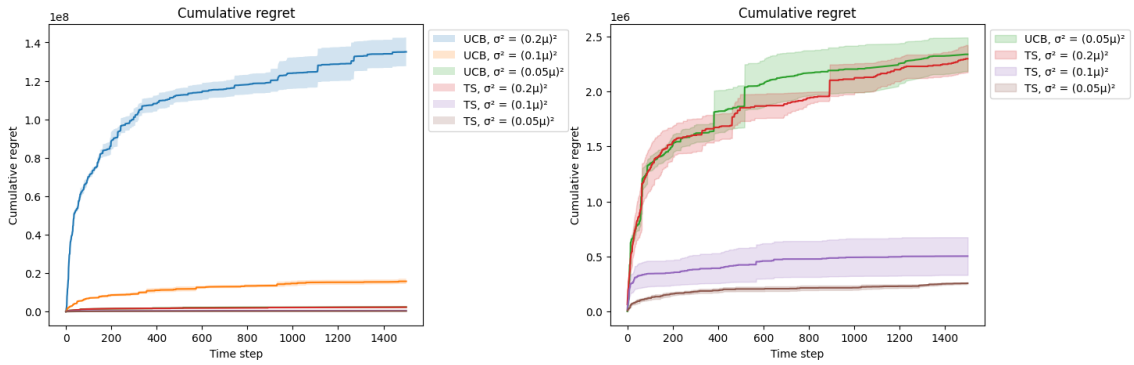


Figure 4.1: Cumulative regret for multi-armed bandit algorithms on a restricted network using UCB and Thompson Sampling as bandit strategies. The values are an average of 3 repetitions, with bands representing ± 1 standard deviation. Different levels of variance are used for both strategies. The second plot shows the same results but without UCB with $\sigma^2_{e,0} = (0.2\mu_{e,0})^2$ and $\sigma^2_{e,0} = (0.1\mu_{e,0})^2$.

cated 4 GB of memory. The run-time was similar for all strategies, and EdgePartition2D was chosen for the following experiments.

EdgePartition2D was then compared to the partitioning strategies using kmeans and GMM clustering. In Figure 4.2 a visualization of the partitioning with the three methods is shown.

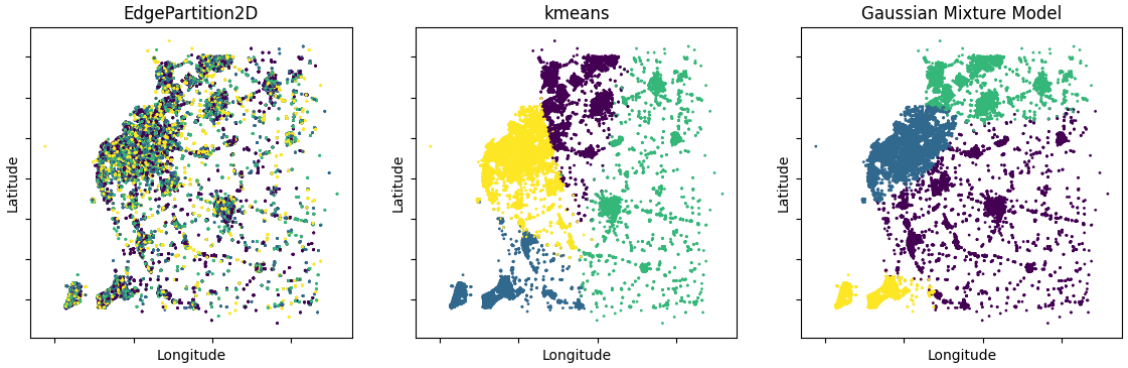


Figure 4.2: Visualization of the different partitioning strategies EdgePartition2D, kmeans clustering and GMM on a network of 40 000 vertices.

While testing different partitioning strategies and their respective run-time, having the same conditions was crucial. The network, source and target vertices and cluster parameters were kept the same, making a comparison possible. Each of the strategies is deterministic, so there are no stochastic elements and therefore no reason to re-partition the graph each iteration.

As seen in Figure 4.3, the run-time for 2 partitions is almost the same for all strategies and the kmeans-strategy has the shortest run-time for 4 partitions. EdgePartition2D has the longest run-time in all experiments and kmeans and GMM has similar run-time for both 8 and 12 partitions. The second plot shows that run-time

increases with network size. The network with 5 000 edges has the shortest run-time with 2 partitions and the networks with 40 000 and 100 000 edges have the shortest run-time with 4 partitions. The third plot shows that the run-time increases as latency is added to the containers.

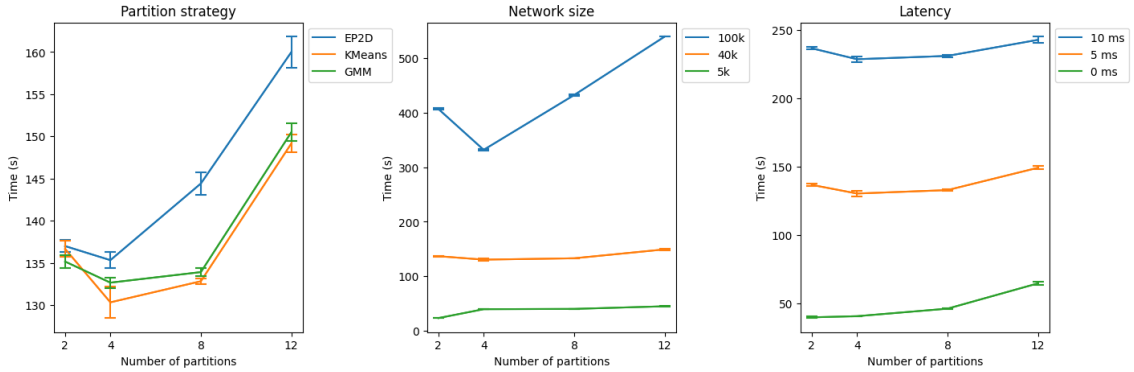


Figure 4.3: Experiments testing partitioning strategy, network size and latency for different number of partitions while measuring the run-time. The run-time is averaged over 6 repetitions with bars representing ±1 standard deviation. In the experiment with partitioning strategy and latency, the map is of Malmö with 40 000 edges. In the experiments with network size, the area of Malmö is extended.

In Figure 4.4, the run-time for KMEANS and GMM partitioning strategies are shown for the cites of Uppsala and Jönköping in the Sweden road network. The Uppsala area consists of 23 000 edges and Jönköping consists of 19 000 edges.

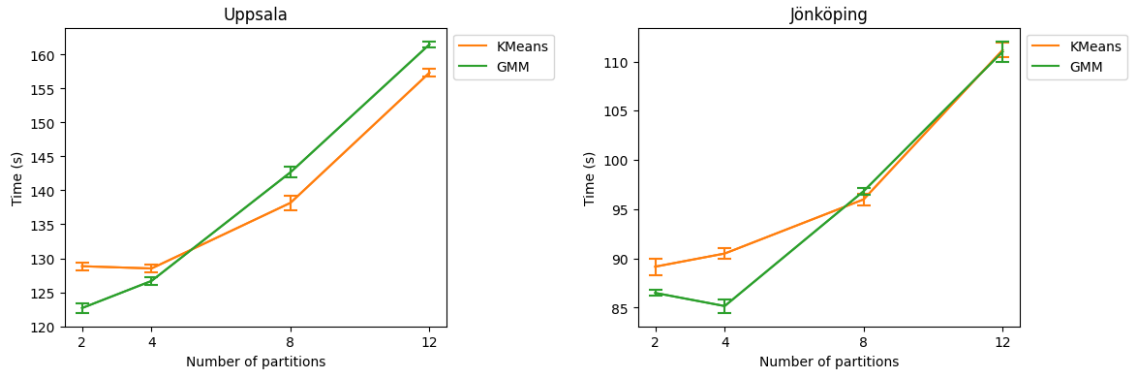


Figure 4.4: The run-time for the partitioning strategies KMEANS and GMM for two different areas of the Sweden road network. Uppsala has 23 000 edges and Jönköping has 19 000 edges. The run-time is averaged over 6 repetitions with bars representing ±1 standard deviation.

In Figure 4.5, the partitioning with KMEANS and GMM from Figure 4.3 and 4.4 is visualized.

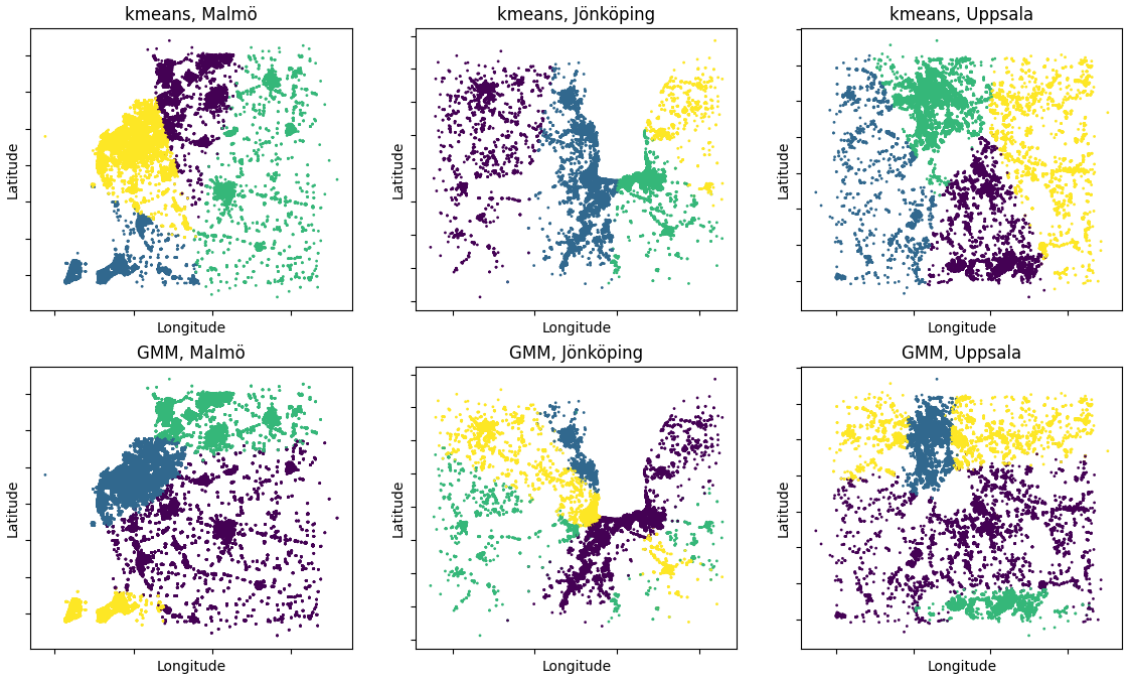


Figure 4.5: Visualization of the partitioning strategies KMEANS clustering and GMM on three different locations in Sweden.

The number of edges in the partitions when partitioning Uppsala and Jönköping with KMEANS and GMM is seen in Figure 4.6.

Since the edges were partitioned, the number of vertices connected to edges belonging to different partitions is a good metric of partition performance. These vertices are called cut vertices and the number of cut vertices when partitioning Uppsala and Jönköping with KMEANS and GMM is seen in Figure 4.7.

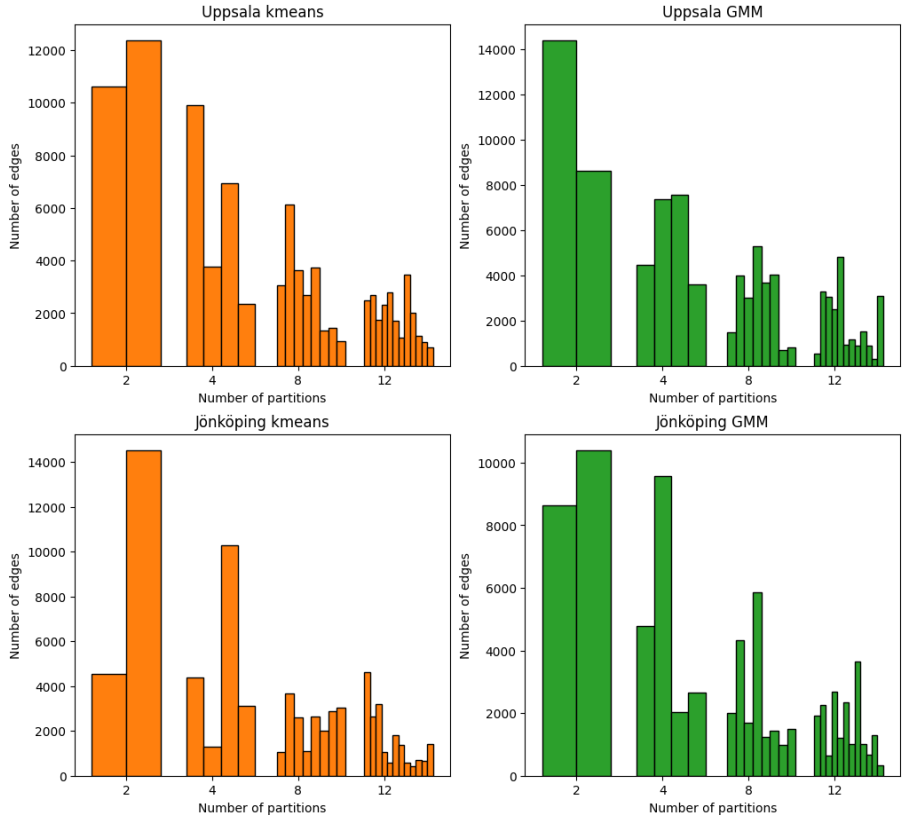


Figure 4.6: The size (number of edges) of the partitions when partitioning Uppsala and Jönköping with KMEANS and GMM.

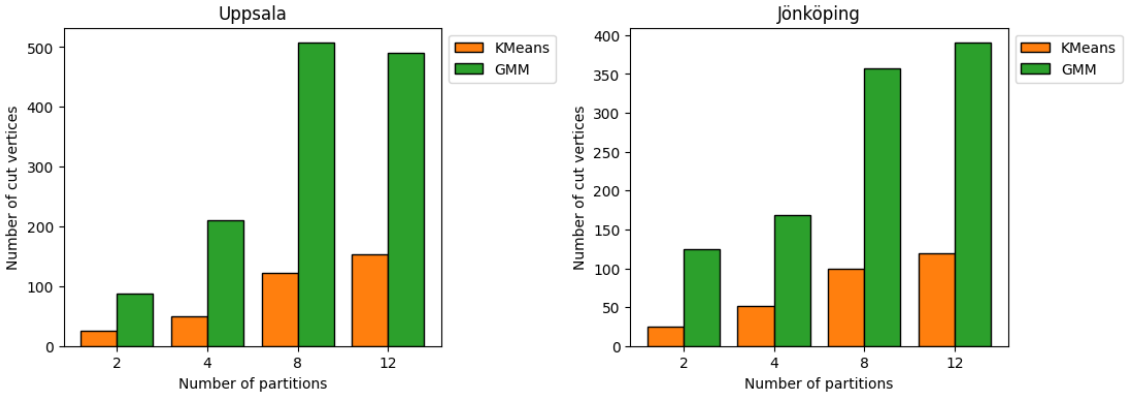


Figure 4.7: The number of cut vertices when partitioning Uppsala and Jönköping with KMEANS and GMM.

5

Discussion and Conclusions

In this chapter, the results are analyzed and discussed. The conclusions are summarized in section 5.2 and the chapter ends with suggestions for future work.

5.1 Discussion

The Discussion section starts with an evaluation of the methods in the project and presents an analysis of the results.

5.1.1 Multi-armed bandit methods

Thompson Sampling performs better in terms of cumulative regret for all levels of the variances, and the difference between Thompson Sampling and UCB becomes more apparent as the variances increase.

This is to be expected due to the nature of UCB. Whenever a path is chosen, each edge of that path is observed, which contributes to reduced uncertainty regarding those edges. Since UCB favour edges with high uncertainty, the algorithm is less inclined to choose that edge again. This can especially be seen in the first 200 time steps when $\sigma^2_{e,0} = (0.2\mu_{e,0})^2$, where the algorithm explores many different paths, accumulating a large amount of regret. This means that UCB explores and observes a larger part of the network, which may be more useful when not only traversing between the same source and target vertices.

Edges with high uncertainty are not favoured to the same degree with Thompson Sampling. A higher posterior uncertainty results in a wider range of edge weights being sampled with Thompson Sampling. As edges are traversed and rewards are observed, the new sampled values will be closer to the expected value of the posterior distribution, since the posterior variance is decreased and exploration becomes less prioritized as the algorithms continue.

A low cumulative regret value is desirable since that means the path with the lowest true expected energy consumption is chosen often, but as other paths are traversed, information of energy consumption is gathered. This information can then be used to correctly choose the least energy consuming path between other vertices, but this

is not taken into consideration when studying the cumulative regret as it is defined here (between two fixed vertices).

5.1.2 Setup

The setup we used to run experiments was a single personal computer with a Docker environment, allowing us to set up multiple Docker containers as virtual machines. Each one was allocated a certain amount of memory and processing power, with simulated latency added to the containers to increase communication time between the containers. This made it possible to simulate a cluster of multiple machines. A drawback with setting up the cluster locally is limitations in memory and the number of processor units, hence the run-times became quite long. It resulted in limitations in how big networks we were able to process. Networks with more than 100 000 edges exceeded the available memory. The number of partitions was also limited since every container represented a machine, and each container needed at least one processor. When trying to run 16 partitions, the memory allocated to each worker was not sufficient, hence 12 partitions was the largest number of workers we were able to run. More memory and processing power would allow experiments with more partitions and larger networks. In [41], a cost model for Pregel was designed, and it was concluded that the execution time depends on the cluster configuration, with the data transfer as an important factor. In our setup, the limited memory is a potential bottleneck for the computation time. For reference, in [41] the authors set up a cluster of 1 master and 5 worker nodes, where each worker had 45 GB of memory, compared to 16 GB in total for all workers in this project.

When looking into the jobs taking place during a run, we could see that a lot of RDD blocks were created during each job. This could be avoided in some cases by caching RDDs to a distributed file system, making the RDDs available for all containers. A full cycle of Pregel consisted of approximately 200+ jobs for a graph with 40 000 edges and 4 partitions. If more RDDs were cached, the number of jobs could potentially have been reduced, which would have decreased the run-time. The reason this was not implemented was due to limited time for the project, and also due to the additional computational resources required. A setup with a distributed file system would need approximately five extra containers, each requiring a share of the available computational resources. With limited resources available, this could potentially affect the performance negatively instead.

5.1.3 Parameters

In this section, we discuss the results from varying the parameters related to the partitioning strategy, network size, latency, and number of partitions.

5.1.3.1 Partition strategy

Looking at Figure 4.2, it is clear that EdgePartition2D is partitioning the edges arbitrarily, even though it is based on the source and target ID. This is because the

IDs do not have any relation to the edge coordinates. KMEANS and GMM exhibit a better ability to partition edges close to each other in the same partition. GMM seems to be slightly better at clustering the edges in the same cities together.

For 2 partitions, the run-time was roughly the same for all strategies. All strategies had their lowest run-time with 4 partitions, after which all run-time increased as the number of partitions increased. The run-time of EDGEPARTITION2D increases significantly more than KMEANS and GMM at 8 and 12 partitions. This may be due to the fact that EDGEPARTITION2D partitions the edges arbitrarily while KMEANS and GMM partitions them based on geographical location. Since cross-partition communication only happens when a vertex crosses partition borders, a strategy that partitions edges based on the edge coordinates is expected to minimize these communications compared to an arbitrary strategy. Since the number of workers is equal to the number of partitions, an increasing number of partitions results in more cross-partitioning communications, especially if the edges are partitioned arbitrarily. Figure 4.7 displays the number of cut vertices for KMEANS and GMM clustering of Uppsala and Jönköping. GMM has approximately three times the number of cut vertices as KMEANS. Still the run-time does not vary significantly. Even though GMM has more cut vertices, they are still quite few. For 4 partitions it has 200 cut vertices, which is approximately 2 % of the total number of vertices.

More experiments with different regions and numbers of partitions would be needed to determine if there is an optimal partitioning strategy. Figure 4.5 visualizes how the two clustering methods differ. KMEANS always creates contiguous cluster while GMM sometimes clusters edges from different parts of the map together. This is expected to increase the run-time. In Figure 4.4 the run-time for the strategies do not differ significantly. GMM is slightly better with 2 and 4 partitions but is afterwards equal, or slightly worse, than KMEANS with 8 and 12 partitions. Looking at the results from the partitioning in Figure 4.3 and 4.4, KMEANS seems to perform better than GMM with 8 and 12 partitions, but it is not possible to see the same trend for 2 and 4 partitions. The fact that GMM cluster edges from different parts of the map together may impact the run-time negatively, which might be why KMEANS performs slightly better in all experiments for 8 and 12 partitions. For 2 and 4 partitions, the results are more inconsistent. The observation that the run-time for 2 partitions differs by approximately 40 seconds between Jönköping and Uppsala and by approximately 5 - 10 seconds between Uppsala and Malmö for KMEANS and GMM, while the difference in edges is 4 000 and 17 000 respectively, may be due to that the clustering performs better for larger cities. It could also be due to the characteristics of the graphs of the different regions.

It would be interesting to investigate how the methods handle a larger number of partitions to see if they differ from each other or not. There is an indication that EDGEPARTITION2D starts to differentiate from KMEANS and GMM after 4 partitions, but more experiments with a larger number of partitions are needed to draw any conclusions. Further, other clustering methods may also be tried, as well as other methods to partition road networks. It would, for example, be interesting to see if geohashing would perform better than clustering. Then it could be possible to have equally sized partitions, in contrast to the clustering methods. This drawback

with the clustering methods results in an unequal workload between the worker nodes, making it difficult to optimize resource allocation. Unequal partition sizes are likely an important factor in why the results for the clustering methods at 2 and 4 partitions vary. As seen in Figure 4.6, the partitioning sizes vary significantly, between partitions and between methods. This may also be an explanation for why EdgePartition2D is not performing significantly worse. Since it is able to create equally sized partitions, the resource allocation is possibly more efficient than with the clustering methods, which may compensate for the increased number of cross-partition communications. Since no other parameters than the sizes of the partitions differ between the EdgePartition2D and the clustering methods, it is likely that this is the reason for only a slight increase in run-time.

5.1.3.2 Network size

For the network with 5 000 edges, the run-time increases with the number of partitions. With 40 000 edges, the best number of partitions is 4 and there is only a slight increase in run-time as the number of partitions increases. With an increase of edges to 100 000, the best number of partitions is still 4, but there is a much larger proportional increase in run-time as the number of partitions increases. As the network increases in size, the importance of partition optimization increases since the difference becomes more apparent. In contrast, as the network decreases in size, the benefits of parallel computing do not outweigh the drawbacks, in terms of cross-partition communications. These results indicate that the optimal number of partitions is dependent on the network size and as the network increases in size, the difference in run-time between the number of partitions becomes greater.

5.1.3.3 Latency

The run-time increased similarly for all levels of latency as the number of partitions increased. The total computing time increased with approximately 100 seconds when 5 milliseconds of latency was added, both from 0 ms to 5 ms and from 5 ms to 10 ms. This suggests that run-time is linearly dependent on the latency added, which remains the same as the number of partitions increases. An increase in partitions creates additional cross-partition communications. With added latency, we expected to see an increased gap between the different series of measurements as the number of partitions increased, but only a small difference appeared. The benefits of distributing computations may be the reason why the increased latency results in a linear increase in run-time as the number of partitions increases.

Even though we specify the latency, the cross-partition communication time fluctuates, where the difference can reach up to 20% of the specified value. The latency is never less than the specified value. Each experiment was repeated and averaged to reduce this stochastic influence. However, this fluctuating latency is likely to exist in a computer cluster with multiple machines as well.

5.1.4 Shortest path algorithm

Our Pregel implementation is based on the GraphX Pregel API, which made it possible to distribute the computations and kept the key function easily implementable.

Using Pregel with an oracle based on a distributed version of Dijkstra's algorithm had limitations, where implementing a condition for ending the search for the target vertex early was not possible. This means that the least energy-consuming path from the source vertex to every other vertex is computed, where an implementation of such a condition may therefore reduce the run-time significantly. The most common way this is implemented in a sequential Dijkstra's algorithm is through exploration of neighbours of the vertex with the current shortest path from the source vertex. This is then continued until the target vertex is reached and the shortest path has therefore been found, assuming there are no negative edge weights. However, this is not easily achievable in a distributed setting, since a lot of information would have to be transferred to the partitions at each step, and as the cross-partition communication cost increases this becomes less efficient.

Pregel has been proven efficient for shortest path problems [9] on networks significantly larger than the ones we use in this project, which justifies the choice of Pregel. The version of Dijkstra's algorithm we use handles more information than the tasks Pregel usually is used for. There are variations of Pregel and other frameworks that enable the distribution of shortest path algorithms [7], which could be investigated in the future.

To be able to run and evaluate the framework, more computations are needed than if this model would have been implemented in a real world scenario. This framework creates underlying reward distributions and simulates observations, which otherwise would be received directly from the environment while traversing an edge in a real-world scenario. To calculate regret, the expected energy consumption and the path with the least expected energy consumption according to the underlying reward distributions has to be calculated, for which we utilize two simultaneous Pregel shortest path algorithms. All the functions that surround the simulations and generate comparable metrics increase the run-time, which together with the limitations in hardware means that the run-times presented most likely are higher than they would be in a real-world implementation, if you disregard the real-world communication latency.

5.2 Conclusions

In this project, we created a distributed framework to find the most energy-efficient path in a road network with stochastic weights, using multi-armed bandit methods. An experimental study was conducted where run-time was investigated in relation to varying number of partitions, partition strategy, latency and network size.

The results show that the framework consistently finds the path with the least expected energy consumption using both Thompson Sampling and Upper Confi-

dence Bound as bandit strategy and is able to do so for different levels of variance. We show that the optimal number of partitions, from a time-efficiency perspective, is dependent on the network size. Dividing the network into partitions according to systematic clustering methods performed better than GraphX's partition strategies, and in our experiments, kmeans performed best. It is, however, from these results not possible to establish an optimal partitioning strategy. Further, having unequal partition sizes seems to have significantly larger impact on the run-time than expected. Increased cross-partition communication time did not have any significant effect on the optimal number of partitions, within the range evaluated.

The project scope was limited by the resources and time available, and available memory was found to be crucial. When larger networks and a larger number of partitions were tested, we ran out of memory. To be able to run larger networks, this is an important factor to take into account when designing the testing environment.

5.3 Future work

To establish a partitioning strategy that efficiently partitions a road network, more experiments are needed where different geographical locations and methods to partition are tested. In this project, the sizes of the partitions in the clustering methods vary. With geohashing, the partitions could be more evenly sized and based on geographical location, by redefining the vertex IDs. This could balance the workload between the worker nodes. If the IDs were dependent on coordinates, it would be interesting to run GraphX's EdgePartition1D and EdgePartition2D strategies and compare them to the cluster methods.

The results show that using 4 partitions or more increases the run-time for all strategies and network sizes. At the same time, the computational resources did not allow networks larger than 100 000 edges. With more resources, larger networks with more partitions could be tried out which may show bigger differences between different partitioning strategies and numbers of partitions. To find an efficient model, more experiments where different combinations of parameters are tested would be needed.

In this project, a distributed file system was not implemented. It would be interesting to implement to see if the run-time may be reduced by continuously caching RDDs, hence reducing the number of jobs performed each iteration.

Since the run-time is heavily dependent on the resources and how they are allocated, analyzing this aspect could give us further insights into the correlation between computational resources and run-time, and how the resource allocation can be optimized. Spark offers a monitoring tool called Spark UI, which presents information about running jobs and breaks down how the job run-time is allocated, e.g. computing time, shuffle write time, etc. Further analysis of this information could be useful for finding bottlenecks in the framework.

Bibliography

[1] E. Commission. "Zero emission vehicles: First fit for 55' deal will end the sale of new co2 emitting cars in europe by 2035." (), [Online]. Available: https://ec.europa.eu/commission/presscorner/detail/en/IP_22_6462.

[2] E. E. Agency. "New registrations of electric vehicles in europe." (), [Online]. Available: https://www.eea.europa.eu/ims/new-registrations-of-electric-vehicles.

[3] L. Noel, G. Zarazua de Rubens, B. K. Sovacool, and J. Kester, "Fear and loathing of electric vehicles: The reactionary rhetoric of range anxiety," *Energy Research Social Science*, vol. 48, pp. 96–107, 2019, ISSN: 2214-6296. DOI: https://doi.org/10.1016/j.erss.2018.10.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214629618304456.

[4] T. Yamashita, K. Izumi, and K. Kurumatani, "Car navigation with route information sharing for improvement of traffic efficiency," in *Proceedings. The 7th International IEEE Conference on Intelligent Transportation Systems (IEEE Cat. No. 04TH8749)*, IEEE, 2004, pp. 465–470.

[5] K. Boriboonsomsin and M. Barth, "Impacts of road grade on fuel consumption and carbon dioxide emissions evidenced by use of advanced navigation systems," *Transportation Research Record*, vol. 2139, no. 1, pp. 21–30, 2009.

[6] N. Åkerblom, Y. Chen, and M. H. Chehreghani, "Online learning of energy consumption for navigation of electric vehicles," *Artificial Intelligence*, vol. 317, p. 103 879, 2023.

[7] D. Zhang, D. Yang, Y. Wang, K.-L. Tan, J. Cao, and H. T. Shen, "Distributed shortest path query processing on dynamic road networks," *The VLDB Journal*, vol. 26, no. 3, pp. 399–419, 2017.

[8] W. Y. H. Adoni, T. Nahhal, B. Aghezzaf, and A. Elbyed, "The mapreduce-based approach to improve the shortest path computation in large-scale road networks: The case of a* algorithm," *Journal of Big Data*, vol. 5, no. 1, pp. 1–24, 2018.

[9] G. Malewicz, M. H. Austern, A. J. Bik, *et al.*, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10, Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 135–146, ISBN: 9781450300322. DOI: 10.1145/1807167.1807184. [Online]. Available: https://doi.org/10.1145/1807167.1807184.

[10] B. Chambers and M. Zaharia, *Spark: The definitive guide: Big data processing made simple*. " O'Reilly Media, Inc.", 2018.

[11] X. Meng, J. Bradley, B. Yavuz, *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

[12] *Graphx programming guide*. [Online]. Available: https://spark.apache.org/docs/latest/graphx-programming-guide.html.

[13] R. Diestel, *Graph Theory* (Graduate Texts in Mathematics). Springer Berlin, Heidelberg, 2017, pp. 2–10, ISBN: 9783662536223. [Online]. Available: https://doi.org/10.1007/978-3-662-53622-3.

[14] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959, ISSN: 0029-599X. DOI: 10.1007/BF01386390. [Online]. Available: https://doi.org/10.1007/BF01386390.

[15] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. DOI: 10.1109/TSSC.1968.300136.

[16] A. Slivkins, *Introduction to multi-armed bandits*, 2019. DOI: 10.48550/ARXIV.1904.07272. [Online]. Available: https://arxiv.org/abs/1904.07272.

[17] D. J. Russo, B. Van Roy, A. Kazerouni, I. Osband, Z. Wen, *et al.*, "A tutorial on thompson sampling," *Foundations and Trendső in Machine Learning*, vol. 11, no. 1, pp. 1–96, 2018.

[18] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, no. 3-4, pp. 285–294, 1933.

[19] O. Chapelle and L. Li, "An empirical evaluation of thompson sampling," *Advances in neural information processing systems*, vol. 24, 2011.

[20] O. Chapelle and L. Li, "An empirical evaluation of thompson sampling," in *Proceedings of the 24th International Conference on Neural Information Processing Systems*, ser. NIPS'11, Granada, Spain: Curran Associates Inc., 2011, pp. 2249–2257, ISBN: 9781618395993.

[21] P. Auer, "Using confidence bounds for exploitation-exploration trade-offs," *J. Mach. Learn. Res.*, vol. 3, no. null, pp. 397–422, Mar. 2003, ISSN: 1532-4435.

[22] E. Kaufmann, O. Cappe, and A. Garivier, "On bayesian upper confidence bounds for bandit problems," in *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics*, N. D. Lawrence and M. Girolami, Eds., ser. Proceedings of Machine Learning Research, vol. 22, La Palma, Canary Islands: PMLR, 21–23 Apr 2012, pp. 592–600. [Online]. Available: https://proceedings.mlr.press/v22/kaufmann12.html.

[23] T. Lattimore and C. Szepesvári, *Bandit algorithms*. Cambridge University Press, 2020.

[24] N. Srinivas, A. Krause, S. M. Kakade, and M. W. Seeger, "Information-theoretic regret bounds for gaussian process optimization in the bandit setting," *IEEE Transactions on Information Theory*, vol. 58, no. 5, pp. 3250–3265, 2012. DOI: 10.1109/TIT.2011.2182033.

[25] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A static performance estimator to guide data partitioning decisions," in *Proceedings of the*

third ACM SIGPLAN symposium on Principles and practice of parallel programming, 1991, pp. 213–223.

[26] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.", 2015.

[27] M. Zaharia, M. Chowdhury, T. Das, *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 15–28.

[28] *Rdd programming guide*. [Online]. Available: `https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations`.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, *et al.*, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[30] *Job scheduling*. [Online]. Available: `https://spark.apache.org/docs/latest/job-scheduling.html#scheduling-within-an-application`.

[31] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 22, no. 8, pp. 888–905, 2000.

[32] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 599–613.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, Ieee, 2010, pp. 1–10.

[34] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[35] *About openstreetmap*. [Online]. Available: `https://www.openstreetmap.org/copyright`.

[36] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta, "An experimental comparison of partitioning strategies in distributed graph processing," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 493–504, Jan. 2017, ISSN: 2150-8097. DOI: `10.14778/3055540.3055543`. [Online]. Available: `https://doi.org/10.14778/3055540.3055543`.

[37] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12, Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 517–528, ISBN: 9781450312479. DOI: `10.1145/2213836.2213895`. [Online]. Available: `https://doi.org/10.1145/2213836.2213895`.

[38] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay, "Clustering large graphs via the singular value decomposition," *Machine learning*, vol. 56, pp. 9–33, 2004.

[39] J. D. Banfield and A. E. Raftery, "Model-based gaussian and non-gaussian clustering," *Biometrics*, vol. 49, no. 3, pp. 803–821, 1993, ISSN: 0006341X,

15410420. [Online]. Available: http://www.jstor.org/stable/2532201 (visited on 05/09/2023).

[40] [Online]. Available: https://hub.docker.com/r/bitnami/spark/.

[41] R. Kumar, A. Abelló, and T. Calders, "Cost model for pregel on graphx," in *Advances in Databases and Information Systems*, M. Kirikova, K. Nørvåg, and G. A. Papadopoulos, Eds., Cham: Springer International Publishing, 2017, pp. 153–166, ISBN: 978-3-319-66917-5.

A

Experimental setup

Experimental setup for all runs, with varying network partitioning strategy, network size and latency for different number of partitions. The number of partitions is equal to the number of workers.

When testing partitioning strategies the settings were:

- **Testing:** Partitioning strategies
  - EdgePartition2D, kmeans clustering, GMM clustering
- **Network size:** 40 000
- **Latency:** 5 ms
- **Workers (*with per-worker memory*):** 2 (*8 GB*), 4 (*4 GB*), 8 (*2 GB*), 12 (*1.3 GB*)

When testing network sizes the settings were:

- **Testing:** Network sizes
  - 5 000, 40 000, 100 000
- **Partitioning strategy:** kmeans clustering
- **Latency:** 5 ms
- **Workers (*with per-worker memory*):** 2 (*8 GB*), 4 (*4 GB*), 8 (*2 GB*), 12 (*1.3 GB*)

When testing latency the settings were:

- **Testing:** Latency
  - 0 ms, 5 ms, 10 ms
- **Partitioning strategy:** kmeans clustering
- **Network size:** 40 000
- **Workers (*with per-worker memory*):** 2 (*8 GB*), 4 (*4 GB*), 8 (*2 GB*), 12 (*1.3 GB*)