On the Optimization of Microservice Deployment through Graph Partitioning

Tyler Ekaireb

June 6, 2022

Abstract

This work explores the use of graph partitioning algorithms to optimize microservice application deployments. Networking overhead presents one of the greatest bottlenecks of service-based architectures. If the application can be modelled as a graph, then graph partitioning algorithms can be exploited to approach a more optimal deployment, collocating services which frequently communicate with one another on the same node, obviating the need for networking between them. In this work, a method to model service-based applications as graphs is outlined. This model is used to construct experiments based on small real-world applications. The experiments are augmented with autogenerated graphs that are meant to represent large-scale applications. It is shown that partitioning techniques from graph theory can be applied effectively and practically to this problem in order to improve application performance in industry settings.

1 Introduction

In order to handle large computational problems, or service the computing needs of many users at once, it is necessary to make use of multiple computers in coordination. Typically, these computers must communicate over a network in order to work in tandem and accomplish their shared goal. Yet, this form of cooperation tends to incur significant overhead. A great many applications would benefit from a reduction of this overhead, necessitating more optimal distributions of the workload in order to minimize network communication. If the workload is modeled as a graph, then graph partitioning can be used to divide the work among nodes while severing as few dependencies as possible.

While the techniques to be discussed can be applied more generally, this work will analyze graph partitioning in the context of microservice deployment. Microservice applications, wherein software components are loosely-coupled and distributed across multiple machines, are quite common in industry. This work aims to gauge the performance of the spectral graph partitioning algorithm in this task. The spectral partitioning algorithm will be compared against another heuristic approach, known as the Kernighan–Lin algorithm, which is often used for partitioning graphs in other contexts.

The contributions of this work include the following:

- A formalization for representing microservices as graphs and optimizing their deployment.
- A framework for mocking microservice applications and simulating automatically generated deployments.
- A set of experiments that evaluate the performance of several graph partitioning algorithms.

2 Background

2.1 Microservices

Microservice architectures are ubiquitous among many of the largest software companies. A microservice application splinters its functionality across many small, self-contained *services*. These services are typically distributed across multiple machines and communicate over a network. There are many advantages to designing software in this fashion. Large applications can be broken down into small services which can be developed and updated independently. Multiple instances of the same service

can be spawned to hot-swap newer versions without downtime, maintain availability in the event of a computer failure, or to accomplish dynamic load balancing. However, microservice applications are subject to significant performance penalties when communicating over the network. It is thus in software engineers' best interest to deploy services with care, distributing them across a set of machines so as to minimize inter-machine communication.

2.2 Graph Partitioning

Graph partitioning is the task of dividing the vertices of a graph into multiple disjoint sets. The applications of this problem range from circuit design to task scheduling. While it is quite straightforward to find a minimum cut to partition a graph, it is generally desirable to impose an additional constraint to find a *balanced* cut. If the task is to distribute a workload across two similar machines, for example, a highly unbalanced partition would lead to an inefficient use of resources.

3 Problem Formulation

In order to make use of graph algorithms, the problem of microservice deployment must be expressed using graphs. A microservice application can be expressed as a set of n services and the connections between them:

$$V = \{s_1, s_2, ..., s_n\}$$

$$E = \binom{V}{2} = \{e : |e| = 2, e \subseteq V\}$$

Thus, the application can be represented as a graph $G = (V, E)$, where V is the set of all services, and E is the set of all edges such that each pair of distinct services is connected by a unique undirected edge. In other words, G is a complete graph where each vertex represents a service in the application.

In this simplified form, there is no distinction between groups of services; every service is connected to every other service by an equivalent edge. If the dependencies between services are known statically, then the edges of G can simply be constructed based on those dependencies, and the unnecessary edges can be eliminated. However, in the general case, the dependencies between services, as well as the degree to which they communicate with one another, are unknown.

While the decision to include an edge between two services may be determined statically to eliminate the unnecessary edges, the model still contains no information about the strength of the connections between services. For example, consider an application $A = \{s_1, s_2, s_3\}$ with edges $E = \{(s_1, s_2), (s_1, s_3)\}$. If s_1 and s_2 share many gigabytes of data in a typical hour of operation, but s_1 and s_3 only exchange a few bytes, then one may say that s_1 and s_2 are more strongly connected. This information is important for the purpose of partitioning the services, because internode communication is typically more expensive than intranode communication due to network delays. Therefore, we would like the model to encode this information as the weight of each edge.

However, it is difficult to determine the weight of an edge, which represents how much data is transferred between two services, via static analysis, because service applications are largely event-driven and thus dependent upon the particular services and functions invoked (and the circumstances of their invocation). To solve this issue, the model defines a *workload* to estimate the weights experimentally. A workload is a set of service invocations that are intended to be representative of typical interactions with the application. Each *request* generates a weighted subgraph, which contains each service that was invoked in order to handle the interaction, and each edge will be weighted by the total amount of data transferred between services:

$$r = \{(a_i, b_i, w_i) : a_i \in A \subseteq V, b_i \in B \subseteq V, w_i \in \mathbb{R} \geq 0\}$$

Each request can also be thought of as a complete graph with all other weights equal to zero. Thus the full workload, which is simply a set of requests to the application, yields a set of graphs whose weights indicate data transfers between services:

$$W \to T = \{t_1, t_2, ..., t_n\}$$

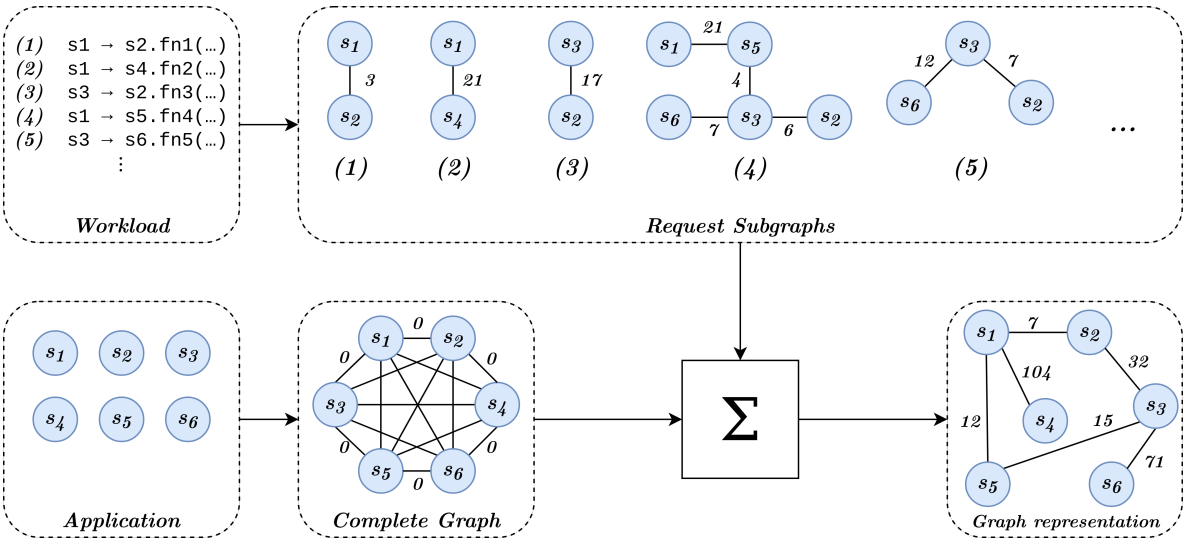


Figure 1: Generating a graph representation of a microservice application

These graphs can be summed together in order to generate the final model representation:

$$G = \sum T = \sum_{t \in T} t$$

This is simply the sum of all edge weights seen across the request subgraphs. In the final graph representation, edges with 0 weight can be ignored. The process of deriving a graph representation from a microservice application can be seen in Figure 1.

4 Algorithms

4.1 Spectral Partitioning

The spectral partitioning algorithm used in this work relies upon the Fiedler vector, as shown in Algorithm 1. The Fiedler vector is the eigenvector corresponding to the second smallest eigenvalue of a matrix. In this case, the matrix is simply the Laplacian of the graph. Once the Fiedler vector of the Laplacian has been calculated, each element is compared to the median of the vector, where each element index corresponds to a vertex in the graph. If the i^{th} entry is less than or equal to the median entry, then the i^{th} vertex will be assigned to partition A; otherwise, the i^{th} vertex will be assigned to partition B:

$$f = \textsc{FiedlerVector}(\mathcal{G})$$

$$A = \{V_i \in V : f_i \leq \textsc{median}(f)\}$$
$$B = \{V_i \in V : f_i > \textsc{median}(f)\}$$

This algorithm is derived from a relaxation of the quadratic optimization problem to find a balanced minimum cut. The simplicity of the algorithm belies the deep mathematical relationship between the graph and its eigenvalues.

4.2 Kernighan–Lin Partitioning

The Kernighan–Lin algorithm attempts to solve the same partitioning problem, namely, minimizing the sum of the weights of the edges cut. However, this algorithm imposes a strict size requirement to divide the vertices into two equal sets, i.e., for a graph with $2n$ nodes, the algorithm yields two sets of size n.

Algorithm 1 SpectralPartition($\mathcal{G}(\mathcal{V}, \mathcal{E})$):

Input: Connected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$
Output: Partitioned vertices of graph $(\mathcal{V}_a, \mathcal{V}_b)$

$f \leftarrow$ FiedlerVector($\mathcal{G}$) ▷ Calculate the fiedler vector of the graph
$c \leftarrow f \leq$ median(f) ▷ c is the result of comparing each element of f against the median element
$A \leftarrow \mathcal{V}[c]$ ▷ A gets all vertices which are less than or equal to the median fiedler vector entry
$B \leftarrow \mathcal{V}[\overline{c}]$ ▷ B gets all vertices which are greater than the median fiedler vector entry

Algorithm 2 KernighanLinPartition($\mathcal{G}(\mathcal{V}, \mathcal{E})$):

Input: Connected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$
Output: Partitioned vertices of graph $(\mathcal{V}_a, \mathcal{V}_b)$

A, B = RandomBalancedPartition($\mathcal{G}$) ▷ Begin with a balanced initial partition of vertices
Compute D values for all $a \in A, , b \in B$
do
  for $i \leftarrow 1$ to $\frac{n}{2}$ **do**
    Find a_i and b_i which maximize $g_i = D_{a_i} + D_{b_i} - 2C_{a_i b_i}$
    Move a_i to B and b_i to A
    Remove a_i and b_i from further consideration in this pass
    Update D values for the elements of $A \setminus \{a_i\}$ and $B \setminus \{b_i\}$
  end for
  Find k that maximizes $g_{max} = \sum_{i=1}^{k} g_i$
  if $g_{max} > 0$ **then**
    Exchange $a_1, a_2, ..., a_k$ with $b_1, b_2, ..., b_k$
  end if
while $g_{max} > 0$

The idea behind the Kernighan-Lin algorithm is to begin with a partition which satisfies the strict size requirement (two equal partitions) and repeatedly swap vertices between the partitions. The procedure is shown in Algorithm 2 [6].

The Kernighan-Lin algorithm begins by randomly dividing the vertices of the graph among two disjoint subsets of equal size, A and B. The cost of a particular partition is the sum of the weights of the edges which cross from A to B. The algorithm greedily pairs vertices from A and B such that swapping them minimizes the cost, provided that neither vertex has been chosen before. The algorithm repeats this process until all vertices have been swapped. The algorithm selects the best partition seen in this iteration, i.e., whichever partition minimizes the cost. While the cost continues to decrease, the algorithm will use the chosen subsets as its initial partition for the next iteration [5]. The algorithm is able to perform one iteration on an n-vertex graph in time $O(n^2 \log n)$ [4].

This algorithm can be readily generalized to an arbitrary number of subsets (k); this would require randomly partitioning the vertices into k initial subsets, then performing the two-subset optimization technique between every pair of subsets. This would be of interest for deployment across more than two nodes. There are also extensions to the Kernighan-Lin algorithm which allow for unequal partitioning (with a known number of vertices per subset). This would be of interest for deployment across heterogeneous devices which each possess varying resources, and thus would be able to support different numbers of services simultaneously (e.g., an asymmetric deployment might be best if the nodes comprise a weaker laptop and a more powerful commercial server). Managing these variables is beyond the scope of this work, so future sections will implicitly refer to the "two equal subsets" variant of the Kernighan-Lin algorithm. However, these questions might be interesting to explore in future work.

5 Implementation

5.1 Overview

The goal of the implemented software is to use a sample workload to approximate an optimal deployment of a microservice application across two nodes. The system simulates the application and logs the data transmitted between services over the course of the workload. This data is used to model a

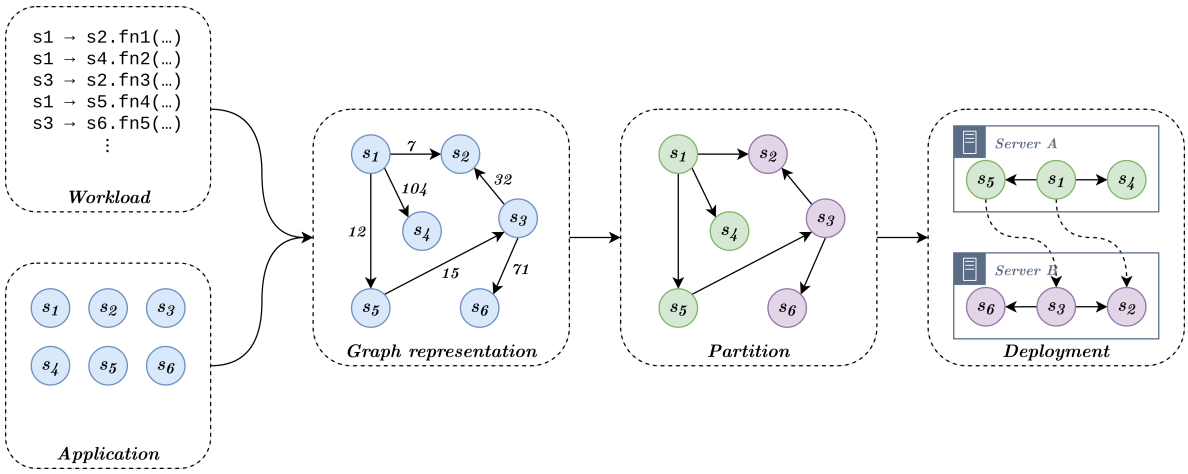


Figure 2: Generating a deployment from a microservice application

weighted graph on which the system executes several graph partitioning algorithms. The partitions are mapped back to services in order to create a deployment. The process is shown in Figure 2.

5.2 Service

This work provides a simple framework to enable the development of microservice applications for the purpose of testing deployment configurations. A `Service` class is provided which allows a name and unique identifier to be assigned to each service. The system also maintains a graph, stored as a Laplacian matrix and/or edge dictionary, to record all communication between services. This graph is fully connected, with a vertex for every service in the application, and edge weights initialized to zero. Whenever a service invokes a function from another service, the sizes of the input parameters and return value are summed together and added to the weight of the edge that connects the two services' vertices in the graph. This is accomplished via a decorator function, which allows additional arbitrary code to be executed upon each service function invocation. Each publicly accessible service function is annotated as a Remote Procedure Call (RPC) with a Python decorator provided by the `Service` class, which handles the data capture and graph updates, as shown in Figures 3 and 4. A microservice application implemented using this framework for testing purposes is described in Section 6.1.

5.3 Graphs

The graphs in this work were represented and manipulated using NetworkX, a Python library for modeling graphs and networks [3]. Some were created by hand based on existing graphs, others were generated randomly using parameterized subroutines, and others still were automatically generated based on recorded communication patterns between services, as mentioned in Section 5.2.

The utility of a graph generated by the system depends upon the degree to which the provided workload is representative of typical interactions with the application in production. The system cannot provide an effective deployment without accurate information on the expected workload. However, representative workloads can be generated rather easily by monitoring the operation of a deployment in the field over a relatively short period of time. The system is already equipped to monitor inter-service traffic, thus it is capable of generating both graphs and workloads, and thereby deployments, on the fly after running in production with an initial naïve deployment. The overhead should be fairly minimal, as the system need only record the size of a few variables and add them to the Laplacian matrix and/or edge dictionary to build the graph iteratively. In future work, perhaps the system could reconfigure deployments in production at regular intervals as the workloads encountered by the application change over time.

5.4 Partitioning

An implementation of the Kernighan-Lin algorithm is provided by the NetworkX library.

```
class Service:

# ...

def rpc(fn):
        def inner(self, sender_id, *args):
            # Get size of messages
            size = sum([sys.getsizeof(arg) for arg in args])
            result = fn(self, *args)
            size += sys.getsizeof(result)

            # Record size of messages (as weighted Laplacian)
            traffic_laplacian[self.id][self.id] += size
            traffic_laplacian[sender_id][sender_id] += size
            traffic_laplacian[self.id][sender_id] -= size
            traffic_laplacian[sender_id][self.id] -= size

            # Record size of messages (as edge dict)
            entry = [self.id, sender_id]
            entry.sort()
            k = str(entry)
            if k in traffic_edges:
                traffic_edges[k] += size
            else:
                traffic_edges[k] = size

            return result

        return inner
```

Figure 3: RPC decorator to capture traffic between services and build a graph representation

```
class ExampleServiceImpl(Service):

    # ...

    @Service.rpc
    def service_fn(self, data):

        # ...

        return result
```

Figure 4: Sample demonstrating usage of RPC decorator to mark a service function

The Python implementation of the Spectral partitioning algorithm used in this work is shown in Figure 5. It is informed by the MATLAB implementation provided by Professor John Gilbert [2].

A subroutine to partition a graph into two equal subgraphs at random was created as well in order to provide a reference for the performance of the two previous algorithms, as shown in Figure 6.

6 Experimental Evaluation

Several sets of tests were devised for this work. A curated set of graphs derived from various microservice applications assesses the efficacy of graph partitioning on real-world problems, and a set of randomly generated graphs simulates much larger applications than could be readily obtained. Finally, an end-to-end test assesses the viability of a system based on this work to be used by engineers in industry settings for automated deployment generation.

```python
import numpy as np
import networkx as nx

def fiedler_bisection(G: nx.Graph):
    fiedler = nx.fiedler_vector(G)
    cmp = fiedler <= np.median(fiedler)
    f_bisection = (set(np.where(cmp)[0]), set(np.where(~cmp)[0]))
    return f_bisection
```

Figure 5: Python implementation of the spectral partitioning algorithm

```python
import numpy as np
import networkx as nx

def random_bisection(G: nx.Graph):
    n = G.number_of_nodes()
    v = np.array(range(n))
    a, b = np.split(v[np.random.permutation(n)], [int(n / 2)])
    return (set(a), set(b))
```

Figure 6: Python function to partition the vertices of a graph equally and at random

6.1 Benchmarks

This work makes use of a variety of benchmarks to assess the utility of graph partitioning under different scenarios. The first set of benchmarks utilizes the Microservice Dependency Graph Dataset, a curated dataset of twenty microservice-based systems [1]. These graphs, provided in SVG format, represent the dependencies between microservices in various applications; an example is shown in Figure 7. The graphs were translated into Python using NetworkX for this work in order for the partitioning algorithms to operate on them programmatically.

The largest graph in the Microservice Dependency Graph Dataset possesses only 25 nodes; this alone is insufficient to generalize the performance of the partitioning algorithms for large-scale deployments. While many prominent companies develop applications and infrastructure with thousands of services, the exact architecture of these systems is not publicly available. To that end, several classes of large graphs were automatically generated for this work: $G_{n,m}$ graphs, where a graph is chosen uniformly at random from the set of all graphs with a specified number of nodes (n) and edges (m); Connected Caveman graphs, where n cliques of size k are formed and connected to adjacent cliques by a single edge; and Relaxed Caveman graphs, also beginning with an initial set of cliques, which are then rewired with probability p to connect different cliques to one another. A sample of each type of randomly generated graph is shown in Figure 8. Integer weights in the range $[1, 100]$ are assigned randomly to each edge. For clarity, the precise weight values are omitted in the figure, but can be approximated visually by the color of each edge: lighter and darker edge colors correspond to lower and higher edge weight values, respectively. $G_{n,m}$ graphs were chosen to represent microservice applications with no regular structure; this is common in small- and medium-sized applications. Caveman graphs were chosen to represent large, decentralized applications, which feature tightly coupled groups of services (cliques) that utilize the functionality of other cliques through designated endpoints. This structure is common in very large applications where multiple teams of developers produce multiple loosely coupled applications that are part of a broader system.

Although these benchmark suites quantify the theoretical effectiveness of the system, they are unable to demonstrate its practicality as a tool. An end-to-end test was devised to fill this gap. A simple microservice application was implemented using the provided utilities. The application allows users to subscribe to various news sources and generate a feed of articles based on their preferences. Given a workload, the system automatically generates a graph representation of the application, performs graph partitioning, and provides a suggested deployment. This end-to-end simulation shows that using graph partitioning for microservice deployment can be done in a nearly automatic fashion, making it feasible to integrate into existing industry applications.

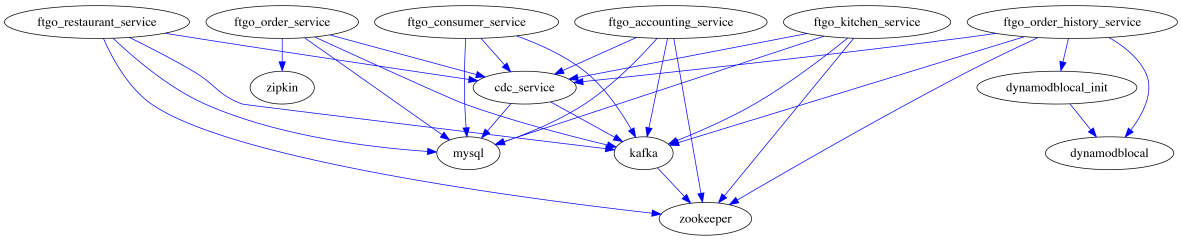
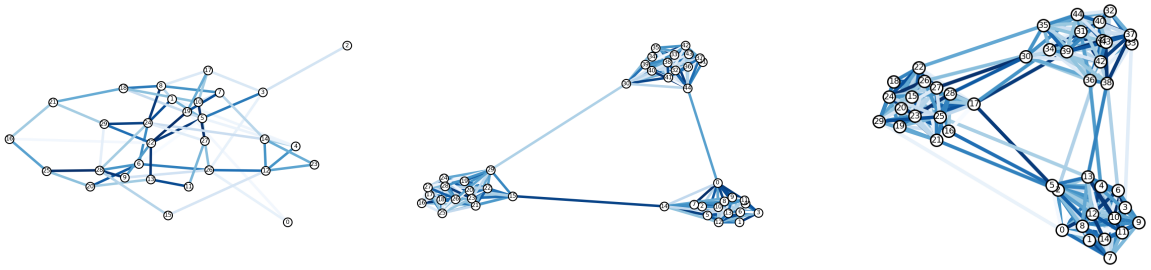


Figure 7: Sample dependency graph provided by the Microservice Dependency Graph Dataset [1]



$G_{n,m}$ graph
$(n = 30, m = 60)$

Connected Caveman graph
$(l = 3, k = 15)$

Relaxed Caveman graph
$(l = 3, k = 15, p = 0.1)$

Figure 8: Samples of randomly generated graphs used for evaluation

6.2 Experimental Results

The partitioning algorithms were first tested on the Microservice Depedency Graph Dataset, whose graphs are unweighted. The goal in this experiment is to minimize the number of severed dependencies, where each severed dependency is given the same cost, as we do not have any networking information to determine the degree of cooperation between services. The results of the two algorithms are compared, along with the results of a completely random partition as a baseline metric, in Figure 9. It is clear that both the Spectral algorithm and the Kernighan-Lin algorithm perform significantly better than a random partition. In this case, the Kernighan-Lin algorithm outperforms the Spectral algorithm slightly.

Partition optimality. Due to the low number of services in applications from the Microservice Dependency Graph Dataset, larger randomly generated graphs were also assessed. In Figure 10, the graphs are unweighted for a more direct comparison with the dataset. The results are very different from the previous experiment. While both algorithms outperform a random partition, the performance of the Spectral algorithm is significantly better than that of the Kernighan-Lin algorithm. While partitioning the largest graph, a Connected Caveman graph with 99,000 edges, the Kernighan-Lin algorithm cut 48,833 edges, which includes almost half of the edges in the graph. For comparison, a random 1:1 partition cut 49,572 edges, so the Kernighan-Lin algorithm performed only 1.5% better than random. For the same graph, the Spectral algorithm cut only 2 edges, which is the lowest possible number of edges, while also achieving the 1:1 split guaranteed by Kernighan-Lin. The results are quite similar when using weighted edges, as shown in Figure 11. In this case, the goal is to minimize the sum of the edge weights, rather than the number of edges cut.

Performance variation by graph type. However, it is clear from the difference in results when operating on the Microservice Dependency Graph Dataset versus the randomly generated graphs that the performance of each algorithm is graph-dependent. Note that both algorithms are heuristic, as finding an optimal partition of non-trivial graphs is prohibitively expensive. As the graphs in the dataset are based on real-world applications, they are difficult to categorize. However, the randomly

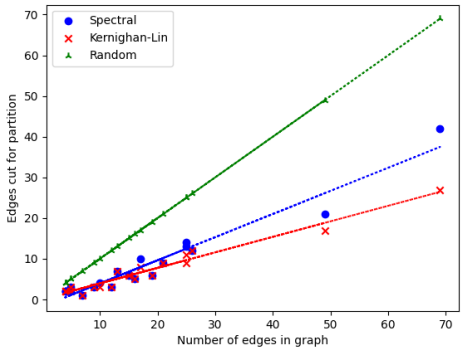


Figure 9: Edges cut to partition graphs from Microservice Dependency Graph Dataset

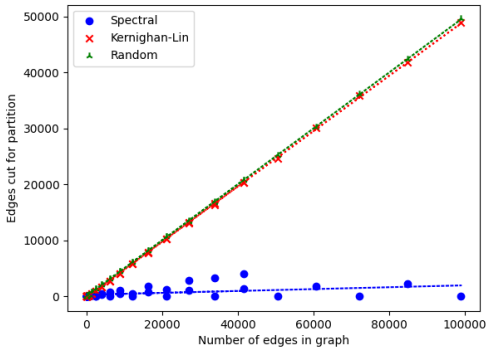


Figure 10: Edges cut to partition randomly generated graphs (unweighted)

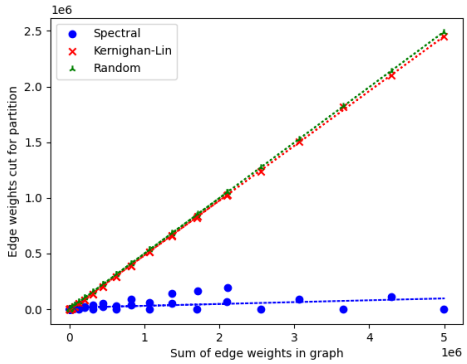
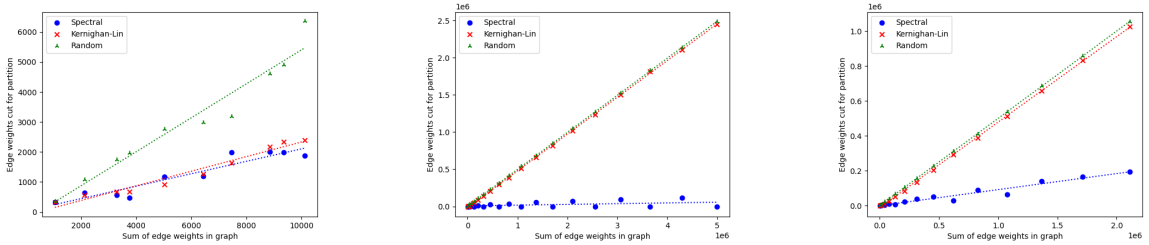


Figure 11: Sum of edge weights cut to partition randomly generated graphs

generated graphs comprise three types of graphs, so the results can be analyzed separately for each category. Consider the performance of each algorithm on each graph type in Figure 12. The performance of the Kernighan-Lin algorithm on various Caveman graphs approaches the performance of a random partition, whereas the Spectral algorithm achieves nearly optimal performance. However, when operating on $G_{n,m}$ graphs, the two algorithms exhibit very similar performance. The trend lines indicate that the Spectral algorithm performs slightly better as the graphs get larger, and the Kernighan-Lin algorithm performs slightly better with smaller graphs. This trend is confirmed by earlier experiments conducted using the Microservice Dependency Graph Dataset, shown in Figure 9, where the Kernighan-Lin algorithm outperforms the Spectral algorithm on small graphs that appear more similar to $G_{n,m}$ graphs than Caveman graphs. For the largest $G_{n,m}$, Connected Caveman, and Relaxed Caveman graphs, the Spectral algorithm cut fewer weights than a random partition, and thus would have reduced network traffic, by factors of 1.25×10^6, 264, and 125, respectively.

End-to-end application. A simple news aggregation application was built in Python for this work. It allows users to subscribe to various news sources or categories of news. Articles from various news sources are aggregated and provided as a personalized "feed" to the user. The application was built using a microservice architecture. It also simulates additional "worker" services to increase complexity, with a total of 15 services. First, a test workload was generated, with several registered users interacting with the application, as shown in Figure 13. Because the application is built using the utilities described earlier in this work, a graph representation of the application based on communication between services is generated automatically. This graph is visualized in Figure 14, with darker edges symbolizing heavier traffic between services. The graph is then provided to the random partitioner, the Spectral partitioner, and the Kernighan-Lin partitioner for testing. Figure 15 shows the resulting partitions of the graph produced by each partitioning algorithm, where all red vertices are grouped together as



$G_{n,m}$ graphs | Connected Caveman graphs | Relaxed Caveman graphs

Figure 12: Edge weights cut to partition each type of randomly generated graph

one subgraph, and all blue vertices are grouped together as another subgraph. The graphs retain the same layout as the initial graph in Figure 14 to facilitate visual comparison. The partitions produced by the Spectral algorithm and the Kernighan-Lin algorithm are somewhat similar, with a difference of 5 vertices. The performance results are summarized in Table 1. Both the Spectral algorithm and the Kernighan-Lin algorithm outperform a random partition significantly, with the Kernighan-Lin algorithm severing less than half the weights cut by the random partition. Recall that in this case, edge weights are representative of communication between services, so collocating services according to the partition provided by the algorithm could reduce network traffic by a factor of 2.2. These results also comport with this work's previous findings regarding the relative performance of the Spectral and Kernighan-Lin algorithms: the Kernighan-Lin algorithm tends to perform better with smaller $G_{n,m}$ graphs, such as this one, whereas the Spectral algorithm tends to perform better with larger graphs and those with more regular structures such as the Caveman families of graphs. This would suggest that practitioners should take the structure of their application's graph into account when selecting a partitioning algorithm, or perhaps use multiple algorithms and choose the best result. Across all tests, including graphs with 2,000 nodes and 99,000 edges, no single execution of a partitioning algorithm exceeded 6 seconds. Thus, in the context of microservice deployment, using both algorithms to generate deployments and compare their results is likely to be highly feasible.

```
usernames = ['Alice', 'Bob', 'Carol', 'Dan', 'Erin', 'Frank']
categories = ['Business', 'Technology', 'US', 'World']

# Build user preferences
for u in usernames:
    # Create user
    end.add_user(u) # Note: `end` is the application endpoint

    # View and subscribe to random categories
    for i in range(3):
        c = random.choice(categories)
        end.get_category(c)
        end.subscribe_category(u, c)

    # Unsubscribe from random category
    end.unsubscribe_category(u, random.choice(categories))

# Get feed
for u in random.sample(usernames, 4):
    end.get_feed(u)
```

Figure 13: Part of a sample workload to interact with the news aggregation application

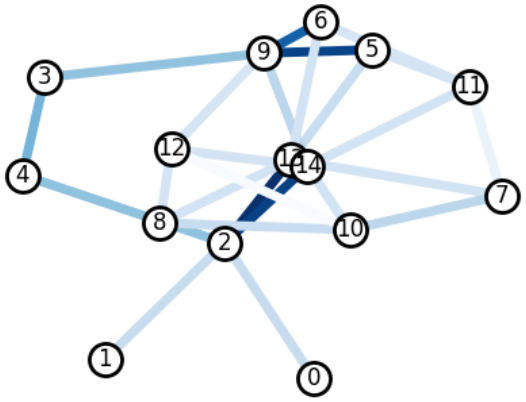
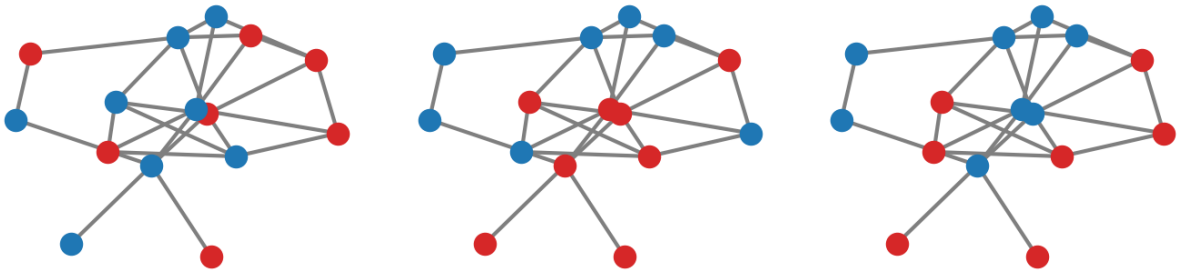


Figure 14: Graph generated based on implemented service and workload



Random Spectral algorithm Kernighan-Lin algorithm

Figure 15: Partitions of the application's graph representation generated by various algorithms

Algorithm	Edge Weights Cut ($\sum$)	Edge Weights Cut (%)	Partition (V_1, V_2)
Random	44,406	60.87%	(7, 8)
Spectral	27,606	37.84%	(7, 8)
Kernighan-Lin	19,969	27.37%	(7, 8)

Table 1: Properties of the partitions generated by various algorithms for the application

7 Conclusion

This work explored the use of graph partitioning to optimize microservice application deployments, and presented a framework to do so automatically. Experimental evaluation demonstrated that given a representative application workload, graph partitioning could reduce inter-nodal network traffic by a factor of 2.2 for a small microservice application. For larger unstructured microservice applications, graph partitioning could theoretically reduce inter-nodal network traffic by a factor of 124. Greater reductions can be achieved for those with certain regular structures, including the largest applications which comprise several thousand services. The results of this work suggest that these techniques could be practical and effective in industry settings for large-scale deployments.

References

[1] CLoud and W. E. (CLOWEE). Microservice dependency graph dataset. https://github.com/clowee/MicroserviceDataset, Aug 2019.

[2] J. Gilbert. Specpart : Spectral partition of a graph. https://github.com/johnrgilbert/S22-graph-laplacians/blob/main/Matlab/Meshpart/specpart.m, Mar 1993.

[3] A. Hagberg, D. Schult, and P. Swart. Networkx. https://networkx.org/, 2004–2022.

[4] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, Sept 1969.

[5] C. Kingsford. Kernighan-lin, graph distance metrics (lecture). https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kernlin.pdf.

[6] C. P. Ravikumar. *Parallel Methods for VLSI Layout Design*. 1995.