Skolkovo Institute of Science and Technology

Master Thesis

DATA-DRIVEN ALGORITHMS FOR SPATIAL NETWORK FLOW PROBLEMS

Student _____ Alvis Logins

Scientific Advisor _____ Panagiotis Karras

Dean of education _____ Clement Fortin

Moscow 2016

Abstract

Given a directed weighted graph, a network flow problem is defined as an optimization problem with an objective function of a form $\sum_E f(e) \cdot w(e) \rightarrow min$, where E is a set of edges of a graph, f is a flow function that associates some flow value to each edge and satisfies conservation constraints, and w is a weight (cost) of that edge; the task is to distribute a flow over network, so as to minimize (or maximize) its total cost, while observing flow conservation and capacity constraints. In a generalization of the problem, called Circulation Problem, a lower bound is added on edge flows. Special cases of such problem include Maximum Flow, Minimum-Cost Maximum Flow, Minimum Cost Flow, Multi-Commodity Flow Problems, as well as Bipartite Matching problem and others. Many applications of a circulation problem deal with spatial data, e.g., traffic analysis, shortest path calculation in transportation networks, optimal package routing in computer networks, and assigning clients to services. The Simplified Incremental Algorithm (SIA) is an algorithm based on incremental subgraph processing and edge pruning that solves the Assignment Problem while taking advantage of spatial indexing, graph density, and certain properties of shortest path computation using the Dijkstra algorithm in bipartite graphs. Previous work has shown that SIA archives a high percentage of pruned edges and, as a result, good performance. For the general circulation problem a Cost-Scaling algorithm is considered as one of the most efficient approaches, yet it does not exploit any pruning.

In this thesis, we first study the SIA algorithm, analyze its implementation details, compare it to algorithms for the general circulation problem applied on the assignment problem, and confirm that SIA outperforms other existing solutions. Then, we design a novel algorithm, S-CSA (Cost Scaling Algorithm optimized for spatial data), that combines an edge-pruning technique similar to the one in SIA with Cost Scaling, in the context of non-bipartite graphs, and investigate its performance. In the process, we adapt a Depth-First implementation of the Cost-Scaling algorithm for use with spatial data and edge pruning. Our experimental study with different types of synthetic graphs and real-data graphs shows

that the performance of S-CSA strongly depends on the initialization of the scaling parameter and the graph topology, while the fraction of pruned edges has the strongest influence on performance; in effect, a simplified version of S-CSA performs well in comparison to both Cost-Scaling and S-CSA with initialization similar to Cost-Scaling. Last, we analyze the performance of the aforementioned algorithms in distributed systems.

Аннотация

Для направленного взвешенного графа задача о нахождении потока определена как оптимизационная задача с целевой функцией вида $\sum_E f(e) \cdot w(e) \rightarrow min$, где E - набор ребер графа, f - функция потока, присваивающая некоторое численное значение потока ребру, и w - вес (стоимость) этого ребра; необходимо распределить поток в сети таким образом, чтобы суммарная стоимость потока была максимизирована или минимизирована и в каждой вершине сумма входящих и выходящих потоков были равны. В наиболее общей формулировке, задаче о циркуляции потока, на ребрах также могут быть определены нижние границы для величины потока. Частными случаями данной задачи являются задачи о максимальном потоке, о потоке минимальной стоимости, о максимальном потоке минимальной стоимости, о назначениях, и другие. Области применения задачи о циркуляции потока часто связаны с пространственными данными, например, анализ дорожного трафика, построение оптимальных путей пакетов данных в компьютерных сетях, и распределние клиентов по точкам обслуживания. Алгоритм SIA (Simplified Incremental Algorithm) предназначен для решения задачи о назначениях, и основан на последовательном рассмотрении подграфов, определенных свойствах алгоритма Дейкстры на двудольных графах, и индексации пространственных данных. По результатам предыдущих работ, SIA позволяет значительно сократить количество используемых при поиске оптимального потока ребер, благодаря чему показывает хорошую производительноть. Для общей задачи, алгоритм масштабирования стоимости потока считается одним из наиболее эффективных, однако не использует возможности отсечения ребер.

В данной работе исследуется алгоритм SIA, детали его реализации и приводится сравнительная характеристика с алгоритмами, предназначенными для более общих проблем, в которой SIA показывает лучший результат. Принципы работы SIA используются для разработки нового алгоритма S-CSA (Cost Scaling Algorithm optimized for spatial data), который бы решал более общую проблему о циркуляции потока и комбинировал в себе методы отсечения ребер и масштабирование стоимости потока.

В процессе разработки, мы исследовали возможности адаптации модификации алгоритма масштабирования стоимости потока для S-CSA, основанной на поиске в глубину. Эксперементальное исследование на различных типах сгенерированных графов и реальных данных показало, что количество отсеченных ребер при выполнении алгоритма S-CSA наиболее существенно влияет на производительность, в то же время сильно зависит от инициализации параметра масшитабирования и топологии графа. В результате, упрощенная версия S-CSA показывает лучший результат по сравнению с алгоритмом масштабирования стоимости потока и S-CSA со схожей инициализацией. Производительность указанных алгоритмов также проанализирована относительно эффективности использования в распределенных системах.

Table of Content

Glossary

Adjacent edges — A pair of edges that share common vertex

Bipartite graph — A graph $G(V,E)$ where nodes belong to two sets V_1, V_2 such that $V_1 \cap V_2 = \emptyset$ and $\forall (v,u) \in E \rightarrow v \in V_1, u \in V_2$ or $u \in V_1, v \in V_2$

Complete Bipartite graph — A bipartite graph, where each node from one subset of nodes is connected with each node of another subset.

Objective function — A function which we aim to maximize or minimize in current optimization problem

Flow — A function that assigns an integer non-negative number to each edge in the directed graph

Flow conservation constraints — A constraint that states that an sum of input flow and output flow should be equal to zero for each node in a graph

Pseudoflow — A flow that does not satisfy flow conservation constraints

Direct edge — An edge of a graph that is given as an input for a network flow problem

Hop — While traversing a graph, a hop is a step - traversing from one node to another along one edge

Inverted edge — An edge that does not exist in initial graph of a network flow problem, but is added to the graph in order to allow the algorithm to cancel flow through direct edge which the inverted edge corresponds to

Definitions and Notations

A list of abbreviations for algorithms:

DF — Depth-First

SSSP — Single Source Shortest Path

SSPA — Successive Shortest Path Algorithm

SIA — Simplified Incremental Algorithm

CSA — Cost-Scaling Algorithm

DF-CSA — Depth-First Cost-Scaling Algorithm

S-CSA — Spatial-oriented Cost-Scaling Algorithm

S-DF-CSA — Spatial-oriented Depth-First Cost-Scaling Algorithm

LDA — Local Dominant Algorithm

Introduction

Given a graph representation of a road map of a city, where a node denotes crossroad and each edge has a length equal to a distance between adjacent crossroads, a common task is to suggest drivers an optimal route to a destination or to simulate traffic over the map in order to analyze, plan and design new transportation infrastructures. Another similar task is to calculate an optimal route of a package in some local area network from one computer to another, minimizing number of hops between routers that the package should pass. These two tasks are special cases of a large class of network flow problems, where a flow should be optimally distributed over a network.

A problem is well-known and a lot of solutions exist, including classical algorithms like Hungarian algorithm, Ford-Fulkerson algorithm, Simplex method, Cost-Scaling algorithm and others, which we will partially cover in this work. Some of them are designed to solve a general Circulation problem, others are more specific and may solve such problems as Minimum-Cost Maximum Flow, Maximum Flow, Bipartite Matching problem and others. Except of managing road traffic and package routing, there are many applications that requires more efficient solutions. For example, computer vision[1], embedded systems [2], car pooling [3].

The current widespread of location-based services and growing interest in distributed systems motivated us to develop a new solution for a Circulation problem that would exploit a spatial properties of input data and minimize the communication cost by reducing the number of active edges. One efficient approach that utilizes the spatial characteristics and the possibility of fast sorting of outgoing edges was introduced in [4]. The algorithm is designed for bipartite matching problem and exploits a possibility to prune high fraction of edges while execution successive shortest path searches. We generalize the algorithm for a circulation problem and apply a cost scaling method in order to deal with weaker pruning threshold. The new approach is then tested on generated and real data.

1 Background

Network flow problems is a wide set of optimization problems of routing a flow in the network while minimizing or maximizing its cost. In real-world applications, a flow can be a road traffic, Internet protocol packets, a water flow or any other entity that should be somehow distributed over a network. Many variations of a problem exist, with slightly different problem statements, objective functions, types of networks and constraints.

In this section, we will formulate Minimum-Cost Circulation problem as the most general Network Flow problem and describe some special cases. Then, we will present a brief overview of existing solutions and describe some of them in details as a preliminary introduction to the novel techniques proposed in this work.

1.1 Problem Statement

The input data for a Minimum Cost Circulation Problem is a graph $G(V,E)$ with a set of nodes V and a set of directed edges E. $(u,v) \in E$ denoted an edge that goes from a vertex u to vertex v. In most of algorithms, it is useful to have a notion of inverted edges. For a given E, we denote $\bar{E}$ as a set of edges, such that $\forall (u,v) \in E \quad \exists (v,u) \in \bar{E}$. Each edge (v,u) from $\bar{E}$ is called an inverted edge of (u,v) from E.

A cost function $c : E \to \mathbb{R}$ and a flow $f : E \cup \bar{E} \to \mathbb{R}$ are functions that assign a real number to each edge. A cost function shows how much advantage or disadvantage does every unit of flow bring if assigned to a particular edge. It can be thought as a weight of an edge, a cost of a flow, or a spatial distance between two nodes. We say that a flow goes from a node u to a node v if there is an edge $(u,v) \in E$ and $f(u,v) > 0$.

Additionally, a flow must satisfy conservation constraints: the total amount of input and output flow for each node must be equal to 0:

$$\forall v \to \sum_{u:(u,v)\in E} f(u,v) - \sum_{w:(v,w)\in E} f(v,w) = 0$$

$\sum_{u:(u,v)\in E} f(u,v)$ is an input flow for the node v and $-\sum_{w:(v,w)\in E} f(v,w)$ is an output flow for the same node.

The flow function must be antisymmetric relating to a set of inverted edges:

$$\forall(v,w) \in E \quad \exists(w,v) \in \bar{E} : f(v,w) = -f(w,v)$$

This property will be later used in most of solutions.

A graph can contain an upper and lower bounds for a flow on each edge. We define $l(u,v)$ and $r(u,v)$ as lower and upper bounds, respectively. A flow that satisfies a non-zero lower bound is called a circulation.

Finally, the Minimum-Cost Circulation problem for a graph $G(V,E)$ is a minimization problem of a form 1.1 with constraints eq.1.2 and eq.1.3.

$$\min_{f} \sum_{(u,v)\in E} f(u,v) \cdot c(u,v) \tag{1.1}$$

$$\sum_{u:(u,v)\in E} f(u,v) - \sum_{w:(v,w)\in E} f(v,w) = 0 \quad \forall v \in V \tag{1.2}$$

$$l(u,v) \leq f(u,v) \leq r(u,v) \quad \forall(u,v) \in E \tag{1.3}$$

In our work, we restrict a cost function $c(u,v)$ and a supply function $sup(v)$ to have only integer non-negative numbers. Additionally, in our experiments we consider the only unit case of all capacities.

For each of problem, maximization and minimization of an objective function are dual operations in each problem. Given a maximization problem, a solution for minimization problem can be derived by making all edge costs negative and incrementing them on a value of a minimum cost in a graph. Given a total flow for modified costs, applying inverse operations leads to a minimized flow. Opposite transformation is similar.

Now, we'll introduce several special cases of the problem and show how they can be solved by finding an optimal circulation.

1.1.1 Minimum Cost Flow problem

In most of applications it is more convenient to formulate a problem in the following way. $s : V \to \mathbb{R}$ is another function that assigns a real number to each node, such that sum of total supplies for all nodes is equal to zero

$$\sum_{v\in V} sup(v) = 0$$

Using this function we can formulate conservation constraint eq.1.2 as eq.1.4.

$$\sum_{u:(u,v)\in E} f(u,v) - \sum_{w:(v,w)\in E} f(v,w) = sup(v) \quad \forall v \in V \tag{1.4}$$

Nodes with positive supply are called sources, and with negative - destinations. A Minimum Cost Circulation problem with sources and destinations is called A Network Flow problem and can be reduced to Minimum Cost Circulation problem in the following manner.

Suppose we have a graph $G(V,E)$ with a set of sources S and set of targets T. Consider a graph $G\prime(V \bigcup p, E \bigcup E_1 \bigcup E_2)$, where p is a node with zero supply, E_1 contain new edges from p to each node in S and E_2 new edges from each node in T to p. All weights of new edges must be zero. If we set $l(u,p) = r(u,p) = s(u)$ and $l(p,w) = r(p,w) = s(w)$ for each $u \in S$ and $w \in T$, and then set $sup(v) = 0$ for all nodes, then this will be a circulation problem and an optimal flow in $G\prime$ will be optimal in G as well, and will satisfy eq.1.4.

1.1.2 Flow Maximization

A Maximum Flow problem is a problem of finding maximum possible flow from a source to a target in a graph with unit weights on edges. This problem can be solved by finding an optimal circulation in the graph by setting all edge costs to 0 and creating the new edge from the target to the source with zero lower bound $l(t,s) = 0$ and infinite upper bound $r(t,s) = \infty$. By setting a cost $c(t,s) = -1$, minimization of circulation cost is equivalent to maximization of flow amount through the newly added edge. Because of conservation constraints, this is also equivalent to maximization of flow amount from the source to the target.

1.1.3 Minimum Cost Maximum Flow problem

Minimum Cost Maximum Flow problem is a problem of finding maximum flow with the smallest possible cost. Unlike other problems considered here, this problem is equivalent to the Circulation problem, so both problems are interchangeable.

If Minimum Cost Maximum Flow problem is given, deriving Circulation Problem goes as follows. First, we must add to a graph an edge (t,s), similarly to the Flow Maximization problem case. Upper bound is also set to infinity $r(t,s) = \infty$, lower bound is zero $l(t,s) = 0$, and a cost is greater than a cost of any other path in the graph: $c(t,s) = -(C+1)n$, where $C = \max_E c(e)$ and n is number of nodes in the graph. If minimum cost circulation is found, then it is both maximal flow and minimum cost. It is maximal because adding one unit of flow through (t,s) edge makes the total cost of a flow smaller than any other path where such unit is not added. So, the amount of flow through (t,s) will be maximum possible and because of conservation constraints, this guarantees for a flow to be maximum flow in the initial graph. Also, minimum circulation contains the minimum flow, because for each maximum flow uses (t,s) edge at the same capacity and, as a result, at the same cost. So, minimum circulation must be also minimum flow in the initial graph without (t,s) edge.

Circulation problem reduces to Minimum Cost Maximum Flow problem as well. To show this, consider any graph for a Circulation problem, add source and target nodes without any edges connected to them. The maximum flow in this network is 0, so a minimum cost flow is actually a minimum cost circulation.

1.1.4 Assignment Problem

An Assignment problem is a problem of assigning clients to services according to preferences of each client and capacity constraints. This can be formulated in terms of graph theory as a bipartite matching problem.

A matching M is a set of pairwise non-adjacent edges in the graph, i.e. $\forall v \in V |\{e \in E | v \in e\}| \leq 1$. A maximal matching is a matching such that any edge in $E \setminus M$ intersects with at least one edge in M. A maximum matching is a matching with a maximum number of edges. A perfect matching is a matching where $\forall v \in V \quad \exists e \in M : v \in e$. Every maximum matching is maximal and every perfect matching is maximum. A bipartite matching is a perfect matching. So, a bipartite matching problem is a problem of maximization or minimization of a total cost of a bipartite

matching in a weighted graph. We want to select a subset M of edges of the initial bipartite graph, such that each node has one outgoing edge in M. In terms of assignment problem, choosing the outgoing edge for a particular node v to a node w that belongs to an opposite subset of nodes means that a client v is **assigned** to a service w. A cost of an edge is equal to a level of desire (or reluctance) of the client to be assigned to the service. This problem can be generalized by adding capacity constraints, that say that a particular node can have only some exact number of outgoing edges (can be more than one) or can have not more than some.

The Assignment problem can be reduced to the Circulation Problem by adding one source node and one target node. Suppose in a bipartite graph we have two subsets of nodes: E_1 and E_2. And for each node a capacity is defined (maximal amount of outgoing edges in M): $cap(v)$. Then, a supply value of new nodes is stated as follows

$$s(source) = -s(target) = \min\{\sum_{v \in E_1} cap(v), \sum_{w \in E_2} cap(w)\}$$

The source node must be connected to each node in E_1, and each node in E_2 must be connected to the target node. Upper and lower bounds of that edges are defined by $cap(v)$ function for each particular node. For example, if a node $v \in E_1$ can be connected to maximum one node from E_2, then upper bound $r(s,v) = 1$. That would mean that no more than one value of flow can pass through that node. Also, for each edge in an initial graph we set upper bound $r(v,w) = 1$. When an optimal circulation found in such extended graph, one unit of a circulation trough an edge between two subsets of nodes is equivalent to choosing the edge as a part of a matching M. No more than one unit can go through one edge, so we can "choose"one edge only once, and all capacity constrains are also satisfied.

In case of strict capacities (for example, when $|E_1| = |E_2|$ and all cap are equal to one, then we can simplify a structure of a graph by defining $sup(v) = cap(v)$ for one set and $sup(w) = -cap(w)$ for another. As a result, we eliminate adding additional source and target nodes, that leads to a better performance of most algorithms. In the case of non-strict capacities, we also

can eliminate additional source and target nodes by adding another node on the smallest size of a bipartite graph, where we could send "extra"flow as suggested in [4].

Note, that the Maximum Matching problem is another problem of finding a perfect matching in a general graph and can not be solved by circulation cost minimization. At the same time, we can imagine an assignment problem in a general graph. Suppose we want to distribute several ongoing cars to available parking places at some moment of time. Such task can be described as a classical bipartite matching problem, where a weight of an edge is the shortest path length on the road map. But, this also can be described as a network flow problem, where we aim to send a unit flow from each car (serving as a source node in a road network) to a parking place (serving as a target node with some capacity). We can "combine"two problem representations and say, that now we are solving assignment problem, but not in a bipartite graph, but in a general graph, that is a road map. This graph has nodes that are either a car or a crossroad. A node that represents a crossroad is a "neutral"node, that has zero supply and that must have an equality between incoming and outgoing edges. Also, such a node can be thought as a node in the partially bipartite graph, that is simultaneously a client and a provider, or that serves as an intermediary between clients and providers. For example, having a computer network system (LAN), a client is a PC, a service is an access point to the Internet, and "neutral"nodes are local routers. This is a classical network flow problem, but still it can be considered as an assignment problem of PCs to access points. We can also apply any restrictions in such a system, for example, a list of restricted connections because of security, or maximal allowed distance between two nodes.

1.2 Solutions of Network Flow problem

In the next few sections, we will give an overview of some existing solutions for Circulation Problem and its special cases. Two algorithms, SIA, and CSA, will be considered in details, since we use them as a basis for our novel approach. Algorithms could be classified by underlying methods

of solution, accuracy, applications, types of input graphs. The solution can be exact ([4]), approximate with some guarantee ([5]), approximate without any guarantee, ϵ-exact or ϵ-scaled ([6]). New approaches are actively developing in recent years both in sequential setting [7] and distributed setting [8, 9]. Parallel implementations can be synchronous and asynchronous. Asynchronous algorithms have a great advantage over synchronous because they don't have any loss of performance caused by a thread waiting and they do not care about workload distribution between threads. The difference is well illustrated in [6].

1.2.1 Exact Algorithms

Exact algorithms are ones that guarantee optimality of an objective function. Most of them are based on iterative shortest path search and optimization of dual variables, that are represented as potential values on nodes.

Single Source Shortest Path Algorithm Single Source Shortest Path problem (SSSP) is, probably, the most common network flow problem. A classical approach to solving SSSP is Dijkstra algorithm [10]. The problem is well studied in the literature and more sophisticated approaches can be used depending on the application. For example, Dial's algorithm is a special case of Dijkstra's algorithm for graphs with integer weights and does not require a priority queue. It is easy-parallelizable, but the scalability of this algorithm depends on the maximum weights of edges and shows an acceptable performance only in case of weights equal to one [11]. If a graph contains large weights, Bellman-Ford is suggested in case of a large number of processes and Delta-Stepping algorithm [12] otherwise. BFS is comparable to Delta-Stepping algorithm but BFS is used in order to simplify description.

Hungarian and Successive Shortest Path Algorithms Originally, Hungarian algorithm [13] was designed to solve assignment problem, however, later the method was generalized to any transportation network by Ford and Fulkerson [14]. It is based on primal-dual method. Given a square matrix of preferences of clients to services, Hungarian algorithm goes

in steps and iteratively decreases row and column values, until zero values in the matrix can be combined with a perfect matching.

Successive Shortest Path Algorithm (SSPA) is a combinatorial adaptation of Hungarian algorithm in terms of bipartite graph [15]. As in Ford-Fulkerson algorithm, it uses potential values of nodes and inverted edges. In this work we will denote potential values as $p(v)$, defining this function as a function that maps each node to a real number:

$$p : V \to \mathbb{R}$$

Inverted edges are defined in the problem statement section. SSPA finds admissible transformations of the graph, based on the shortest path from a non-assigned client to available service. After the shortest path is found, all edges are inverted along the path and potentials raised. Evaluating m shortest path searches, where m is a number of clients, SSPA finds an optimal assignment.

Blossom and Cost Scaling Algorithms

An overview of the evolution of Hungarian-based algorithms is presented in [7], as well as a new algorithm that is based on blossoms and linear optimization with both approximate and exact versions for both bipartite matching and maxflow problems. A blossom is a set of vertices that are connected in a circle and a subset of edges of a blossom is in maximal matching. This means, there is a subset of edges that contains exactly one edge for each vertex. These blossoms are used in order to iteratively build alternating paths, that is a sequence of vertices that are connected and forms maximal matching, together with two desolated vertices, one in the beginning of the sequence and one in the end. Alternating paths then iteratively alternate in order to include desolated vertices in maximal matching.

Originally, the concept of blossoms was introduced by Edmonds [16] and then improved by Kolmogorov [1], which developed a way to reduce a problem to a linear optimization and find augmenting paths in a way that maximized (minimizes) the total sum of weights in a resulting subset of edges. He also developed a well-written library in C++, that currently is the best

non-distributed software for solving maxflow problem in general graphs, up to the best of our knowledge.

Later, Goldberg and Tarjan presented an update to their well-known push-relabel algorithm [17], presented more precise theoretical complexity estimates and better methods to solve subroutines of the algorithm. In the context of previously mentioned works, Goldberg showed that the approach of Duan, which in its turn took several ideas of Kolmogorov, can be represented in a significantly much simpler way, if expressed in terms of the framework presented in [18]. Moreover, this simplification leads to a better upper bound.

BlossomV One of the best novels approaches based on the blossom concept is provided by [1]. BlossomV solves the general matching problem, that belongs to a convex optimization problems. Therefore, it can be solved by according software and methods like primal simplex methods, relaxation methods, dual simplex methods can be applied. The main idea of the algorithm is to make an iterative reduction of a graph by finding blossoms - cycles of alternating paths in a graph. Then, the problem is reduced to convex optimization. In the current thesis, we compare the performance of this algorithm on a special case of bipartite matching with other approaches.

Primal Simplex Method As we can see from the problem statement, Circulation problem is a special case of the linear optimization problem. An efficient adaptation of linear optimization for graphs is presented by [19] and is also considered as one of the most efficient approaches nowadays, together with Cost-Scaling algorithm [20].

1.2.2 Distributed Algorithms

In the distributed setting, the development of algorithms for solving mentioned problems seems not so active as in the sequential case. Most works are concentrated on finding an approximate solution. Some classic combinatorial optimization problems such as matching, coloring, dominating set, or approximations can be solved using small (i.e., polylogarithmic) local communication [21]. Here we mention some of the complexity results that might be relevant to the maxflow problem.

The lower bound for distributed system for matching problem in general graphs is $O(\log^2 n)$ with constant approximation ratio. Lower bound shown by [22] is $\Omega(\log \Delta / \log \log \Delta + \sqrt{\log n / \log \log n})$.

Tight lower bound on the communication complexity of approximate maximum matching in bipartite graphs was shown by [9] and is equal to $\Omega(\alpha^2 kn)$. It is tight up to poly-logarithmic factors. In related works, they mention algorithm (Lotker) with $O(\log n)$ rounds.

In asynchronious networks [23] presents a shortest path algorithm that converges in a finite number of steps.

The most interesting result in finding approximate maximum flow in a graph is presented by Ghaffari et al. [8]. They present a near-optimal distributed algorithm for $(1 + O(1))$-approximation of maximum flow in undirected network using $(D + \sqrt{n})n^{o(1)}$ communication rounds. Their approach is based on congestion minimization with gradient descent, where congestion of an edge is defined as used capacity of an edge divided by its total capacity.

The notable thing about this work is that the exact calculation of an objective function, which they derivate in presented framework, is too difficult to solve exactly, so they compare various methods to build a congestion approximator R - a matrix that would reduce a constrained optimization problem to an unconstrained, that then could be easily solved by gradient descent. The basis for building R is a generation of a sample of random graphs over the network, where each graph contains some subset of edges. This concept was originally called as building graph spanners [24] and a motivation is to build a simpler representative of a network, that hold some specific properties, and then by applying an algorithm to this simplified set of representatives, derive a desired property of the whole network.

This approach became very popular after Bartal introduced hierarchal decomposition of trees (HDT) [24] and showed that for a random sample of trees holds a remarkable property: expected value of the ratio of distance in HST and in original graph is not larger than some threshold. This result is very powerful as it allows for many problems to first solve the tree instance of the problem and then to transfer this solution to the original graph while

paying only a factor f in the performance guarantee [25]. We can imagine an example of exploiting the properties of a tree, when a tree is colored in almost constant time [26] and then chromatic scheduling [27] used for distributed calculations with dependences.

Recent works in distributed algorithm for exact maximum flow problem show $O(n^2)$ time and $O(n^2m)$ message complexities [28], that is comparable to result in we obtained for current algorithm if the diameter is small.

1.2.3 Parallelization

A good example of a work related to parallelization is presented by [29]. Authors use Local Dominant algorithm (with 1/2-approximation guarantee) which is a modification of [30] and [31]. They provide a solution for multicore (Intel Nehalem and AMD Magny-Cours), manycore (Nvidia Tesla and Nvidia Fermi), and massively multithreaded (Cray XMT) platforms with strong emphasis on Cray XMT platform. However, they use native LD algorithm for multicore and GPU, but they modify it by embedding some atomic specific commands for Clay XMT. Probably, the performance of GPU can be improved by modifying the algorithm and considering the specifications of GPU architecture.

LD algorithm is simple and has a straightforward parallelized and distributed versions and can be simply used for architecture comparison.

Another way of parallelization is presented in [32]. Authors present parallel Hungarian algorithm, which has a pretty matrix representation. So, they reduce the algorithm to a set of matrix operations which can be done in parallel.

1.2.4 Dynamic Algorithms

Another class of problems is updates and dynamic solutions. This case has a wide spectrum of applications, especially in the case of approximate solutions.

One of the solutions is presented by [4]. It considers a case for customer updates in bipartite matching. The solution is based on two observations. First is that if we update distance function of some customer, then we can update potentials as follows: $p(t) = \max_{e(q_k,t)\in E \wedge w(q_k,t)<0}\{-dist(q_k,t) + p(q_k)\}$, where t is a moving customer. This update leads to the elimination of negative costs and does allow the algorithm to use results calculated for non-updated customers, i.e. to continue running. The second observation is that if we use a subgraph E_{sub}, after an update we can recalculate edges in E_{sub} by using the distance threshold: all edges that are smaller than the largest edge added so far, must be added to E_sub after an update. Additionally, we fulfill (invert) those edges, which have negative reduced cost.

The technique of inverting edges with a negative reduced cost can be widely used in any algorithm that uses the notion of reduced cost. For example, in the CSA after new edge addition or any node potential change we can rearrange all adjacent edges and this will preserve all algorithm invariants, i.e. all flows will still be ϵ-optimal because each affected edge has non-negative reduced cost. However, this would create a pseudoflow, so flow refinement must be performed after an update.

Caching can be used in order to apply current solution for real-world data [33].

1.2.5 Approximate Algorithms

There are several basic approaches for object function approximation. Since the problem can be solved exactly, the usually approximate algorithm is significatly faster than exact solutions. In [4] approximation is done by grouping close objects in space, where closeness in defined by an approximation parameter and found using R-tree and Hilbert curve.

Local Dominant Algorithm Local Dominant Algorithm (LDA) is an efficient approximate algorithm for solving the matching problem based on the notion of dominance [31, 34]. A locally dominant edge is at least as heavy as any other edge incident its end points. Each node can set a pointer to its heaviest neighbor. If two nodes point to each other, then the edge is

locally dominant. The algorithm goes as follows. First, we set for each node a candidate node, i.e. a pointer to the heaviest neighbor. Then, if a pair of nodes is found where both are nodes are stated as heaviest, then the edge between them is locally dominant. Locally dominant edge is saved and both nodes are enqueued. The pseudocode is shown on Algorithm 1. A list of locally dominant edges is saved in *mate* property of each node. The algorithm has both straightforward sequential and parallel implementations.

Algorithm 1 Local Dominant Algorithm

```
1:  Input: G(V,E)
2:  Q = new queue
3:  for all v ∈ V do
4:      v.candidate = arg min_{w:(v,w)∈E} c(v,w)
5:      w = v.candidate
6:      if w.candidate == v then
7:          v.mate = w, w.mate = v
8:          Q.push(v),Q.push(w)
9:  while not Q.empty do
10:     v = Q.pop
11:     v.candidate = arg min_{w:(v,w)∈E} c(v,w)
12:     w = v.candidate
13:     if w.candidate == v then
14:         v.mate = w, w.mate = v
15:         Q.push(v),Q.push(w)
```

Some papers provide an approximation guarantee. For example, [29] proposes an algorithm that gives a solution for a maximization problem where a total resulting sum of weights is not less than half of total sum of the exact solution. The half-approximation guarantee in [29] and similar works is based on the article [31], where a variation of LDA, called LAM, is described. Lemma 3 and Lemma 4 in [31] is a proof of the approximation quality. Lemma 1 in this work refers to the Lemma 3 in [31]. In fact, it described the principle of assignment in LDA. The proof just folllow the pseudocode of the algorithm.

Lemma 1. *Algorithm LAM starts with an empty matching and an edge (a,b) is only added is a and b are free and neither a nor b are adjacent to a free vertex with an edge of higher weight than (a,b).*

We provide a full version of the second lemma as Lemma 2 in this work. the approach is not "symmetric"when we try to solve minimization problem. Please, refer to [31] for full original versions of lemmas.

Lemma 2. *Algorithm LAM computes a matching M_{LAM} with at least $\frac{1}{2}$ of the edge weight of a maximum weight matching M_{MWM}, where MWM is minimum weight matching.*

Proof. Lev V_{LAM} be the matching vertices in M_{LAM} and V_{MWM} in M_{MWM}. Throughout the algorithm the following holds

$$W(M_{LAM}) \geqslant \frac{1}{2}W(\{\{\{u,v\} \in M_{MWM} | u \in V_{LAM} \vee v \in V_{LAM}\})$$

When $M_{LAM} = \emptyset$, adding (a,b) to M_{LAM} increases $W(M_{LAM})$ by $w(a,b)$, but also the right side may increase. If $(a,b) \in M_{MWM}$, the right hand side only increases by $\frac{1}{2}w(a,b)$. Otherwise, let (a,c),$(b,d) \in M_{MWM}$ be the possible edges adjacent to (a,b). The choise of matching edge (a,b) excluded the possible choise of (a,c) and (b,d) throughout the rest of the algorithm. There are the only two edges by which the subset of M_{MWM} may increase, i.e. the right hand side may only increase by $\frac{1}{2}(w(a,c) + w(b,d))$. If $c \in V_{LAM} (d \in V_{LAM})$ before wi add edge (a,b), then (a,c) $((b,d))$ is already in the subset of M_{MWM}. If $c \notin V_{LAM}$ $(d \notin V_{LAM})$, i.e. c (d) is free, Lemma 1 insures that $w(a,b) \geqslant w(a,c)$ $(w(a,b) \geqslant w(b,d))$. Therefore, the value on the right hand side cannot increase by more than $w(a,b)$. At the end, LAM terminates with a maximal matching and the following holds

$$W(M_{LAM}) \geqslant \frac{1}{2}W(\{\{\{u,v\} \in M_{MWM} | u \in V_{LAM} \vee v \in V_{LAM}\}) = \frac{1}{2}W(M_{MWM})$$

□

Lemma 1 remains valid for a minimum cost problem: an edge $\{a,b\}$ is only added if a and b are free and neither a nor b are adjacent to a free vertex with an edge of **smaller** weight than $\{a,b\}$. The proof has direct

inverted version. But Lemma 2 does not. The equation that must be proved for minimum-weight version is the following:

$$W(M_{LAM}) \geqslant 2W(\{\{\{u,v\} \in M_{MWM} | u \in V_{LAM} \vee v \in V_{LAM}\})$$

After adding an edge $\{a,b\}$ to M_{LAM}, $W(M_{LAM})$ increases by $w(\{a,b\})$ and the right hand side may increase. The key point is that right hand **may** increase, but may not. If it increases, then the equation holds because of the inverted version of Lemma 3. But, if both edges c and d are already added, then right hand side is not increased at all and the equation is not valid any more.

At the same time, in a full graph (that is equivalent to the spatial data) and with random weights the optimal minimum sum of weights decreases with the number of nodes. This is an experimental result obtained in the experiments described later. The intuition is that if a number of nodes is big enough, an exact algorithm can find a pair of matching with minimum weight for almost every node. So, if a number of nodes is very big, then each node is matched using an edge of zero-weight and the total sum is almost zero. In this situation, every mistake of an approximate algorithm could increase the total sum (final result) many times ($\gtrsim 10^4$). This could be a good motivation for using an exact algorithm for a min-weight matching problem instead of an approximate one, even if the latter is faster.

1.3 Simplified Incremental Algorithm

Simplified Incremental Algorithm (SIA) [4] is an algorithm for solving a bipartite matching problem with capacity constraints. Authors propose several improvements of the SSPA (Successive Shortest Path Algorithm) that lead to the great increase in the performance. SIA uses concepts of inverted edges, potentials, iterative shortest path search, that makes it similar to such algorithms as Ford-Fulkerson, Hungarian, Cost-Scaling and others. Algorithm 2 illustrates the complete version of SIA. We used original notations of authors. $q_i \in Q$, $p_i \in P$ are nodes from the first and the second subsets of V of a bipartite graph $G(V,E)$ accordingly. τ is a potential. $q.\alpha$, $p.\alpha$ are minimum distance values that were calculated during the shortest

path search. $dist()$ is an edge cost function. e refers to an edge. $v.prev$ is the previous node in the shortest path of a node v. w is a reduced cost of an edge. v_{min} is a final node in the shortest path. v_{min} is always a non-full p_i.

The distinctive feature of SIA is its pruning technique of edges. The algorithm tries to find the shortest path from the source q_i to a non-full target p in a subset of edges, called E_{sub}. This subset is initialized as the empty set (line 2). There are γ iterations in the algorithm (line 3), where γ is a number of customers, i.e. nodes in the first subset of V. At each iteration, a shortest path between non-full nodes of opposite subsets of a bipartite graph is found. Each shortest path is found using only edges in E_{sub}. Lines 10-21 show the shortest path search. A threshold (line 10) guarantees that the shortest path that was found in E_{sub} using Dijkstra (line 13) is the shortest in E as well. Lines 23-25 reverses all edges along the shortest path. Lines 26-32 are responsible for updating potentials in the graph. Maximum value of potential value τ'_{max} is maintained on line 33.

The pruning goes as follows. At each iteration of the algorithm a heap H (line 4) stores values $q.\alpha + dist(q_k,p_m)$, i.e. a minimum distance to the enheaped node plus a cost of an edge to the next nearest neighbor of the node. New values are enheaped each time a shortest path algorithm discovers a new node (line 21) or when a minimum distance to some old node becomes updated (line 21). Using the heap, E_{sub} becomes distance-bounded, i.e. after the shortest path search any edge that is not in E_{sub}, but still belongs to the graph, is not smaller than some value. Authors [4] prove that such threshold guarantees that the shortest path in E_{sub} is also the shortest in E by showing that any path with any edge $e \in E \setminus E_{sub}$ is longer than current shortest path. Intuitively, the threshold on line 10 shows that we should add more edges in E_{sub} while current minimum distance the target is larger than a minimum reduced cost of any "relevant"edge, where "relevant"is such edge that can influence the shortest path, and reduced cost is a positive cost w that is guaranteed to be smaller than total cost of the shortest path.

Additionally to SIA, authors suggest Path Update Algorithm (PUA), that is the extension of SIA. It allows for every Dijkstra execution to partially reuse results of previous executions. Algorithm 3 illustrates PUA. H_d refers to

the heap of the last Dijkstra execution. The intuition behind the algorithm is the following. After each insertion of new edge, we continue running Dijkstra, which was terminated in the previous step because of lack of nodes or infeasibility of a solution. Before doing that, we update minimum distances of all nodes which were affected by the newly inserted edge. Another heap, called H_f, is used for that reason. In fact, this is another instance of Dijkstra, which starts from the inserted nodes and finishes on any node that exists in a heap H_d, remained from the previous iteration (before new edge insertion). From now on we will refer to SIA with PUA extension as just SIA.

The success of the algorithm is motivated by the tightness of the threshold. In bipartite graphs, the total length of the path tends to be small. In particular, in a complete bipartite graph, the length is no longer than 1, even if total hops the algorithms does during the graph traversal in Dijkstra algorithm can be much larger. Intuition is that each second hop in any shortest path found by Dijkstra is equivalent to canceling some assignment, because each second hop must go through inverted edge that has a negative cost and indicates a possibility of canceling flow through a direct edge.

1.4 Network flow algorithms for spatial data

SIA takes advantage out of a possibility to incrementally retrieve next nearest neighbors of each node. In case of large graph this is a significant benefit in terms of both memory and time. In the next chapter, we will discuss the design of a new approach that also proceeds edges for each node in increasing order of their length. Although the requirement of a list of outgoing edges being sorted can be achieved by graph preprocessing, we focus on the spatial data as an application of new algorithms. Spatial data management is an actively developing topic and does not require additional preprocessing of data. In this section we briefly discuss the possible state-of-the-art approaches to index spatial data and incrementally retrieve nearest neighbors.

Spatial indexing is a common approach to deal with big spatial data. It is used in Spatial Joins [35], Spatio-Textual Joins [36], Road Map Shortest Path Queries [37, 38]. Caching can be applied for spatial indexing [39]. The dominant indexing method for spatial data is R-tree [40, 35, 36]. It uses

Algorithm 2 Simplified Incremental Algorithm (SIA)

1: $\tau_{max} := 0$, $E_{sub} := \varnothing$
2: **for** $loop := 1$ to γ **do**
3: $H :=$ new min-heap; set $v.\alpha := \infty \, \forall v \in E_{sub}$
4: select a non-full $q_i \in Q$ in round-robin fashion
5: $q_i.\alpha := 0$, $v_{min}.\alpha := \infty$
6: $p_j :=$ first NN of q_i in P
7: insert $\langle e(q_i,p_j), dist(q_i,p_j) \rangle$ into H
8: **while** $v_{min}.\alpha > TopKey(H) - \tau'_{max}$ **do**
9: de-heap $\langle e(q_k,p_j), key \rangle$ from H
10: insert $e(q_k,p_j)$ into E_{sub}
11: $v_{min} :=$ Dijkstra(Q,P,E_{sub})
12: **for all** visited $q \in Q$ **do**
13: **if** q is not in H **then**
14: $p_j :=$ get next NN of q in P
15: insert $\langle e(q,p_j), q.\alpha + dist(q,p_j) \rangle$ into H
16: **if** $q.\alpha$ changed in Line 13 **then**
17: update $q.\alpha$ in H
18: $p_m :=$ next NN of q_k in P
19: insert $\langle e(q_k,p_m), q.\alpha + dist(q_k,p_m) \rangle$ into H
20: $v := v_{min}$
21: **while** $v.prev \neq \varnothing$ **do**
22: reverse $e(v,v.prev)$ in E
23: $v := v.prev$
24: **for all** visited nodes v_i **do**
25: $v_i.\tau := v_i.\tau - v_i.\alpha + v_{min}.\alpha$
26: **for all** edges $e(v_i,v_j)$ incident to v_i **do**
27: **if** $v_i \in Q \cup \{s\}$ **then**
28: $w(v_i,v_j) := dist(v_i,v_j) - v_i.\tau + v_j.\tau$
29: **if** $v_i \in P$ **then**
30: $w(v_i,v_j) := -dist(v_i,v_j) - v_i.\tau + v_j.\tau$
31: $\tau'_{max} = \max\{v.\tau | v \in Q \wedge v.\alpha \leqslant TopKey(H)\}$

Algorithm 3 Path Update Algorithm

```
1:  H_f := new min-heap
2:  insert ⟨q, q.α⟩ into H_f
3:  while H_f is not empty do
4:      de-heap top node v_i (with the lowest v_i.α value) from H_f
5:      for all edges e(v_i,v_j) ∈ E_sub outgoing from v_i do
6:          if v_j.α > v_i.α + w(v_i,v_j) then
7:              v_j.α := v_i.α + w(u,v); v_j.prev := v_i
8:              if v_j ∈ H_d then
9:                  update v_j.α in H_d
10:             else
11:                 if v_j ∈ H_f then
12:                     update v_j.α in H_f
13:                 else
14:                     insert ⟨v_j, v_j.α⟩ into H_f
```

Minimum Bounding Boxes (MBR) for spatial object indexing. Having MBR for each object, R-tree builds a tree structure, where leafs are MBRs that contain objects of arbitrary form and nodes are MBRs that contain MBRs of their children. Figure 1.1 shows an example of an R-tree. The bottom level (red) contains leafs with objects. The performance of R-tree depends on the node splitting strategy, such as linear and quadratic algorithms [41, 42]. Several R-tree modifications were proposed in the literature, such as R*-tree and R^+-tree [35]. Some of them may use bulkloading, i.e. building a tree starting from the leafs instead of incremental object insertion. For example, Hilbert curve can be used to define a linear order for spatial objects and group spatial data according to this order. This will put spatially close objects to one group with a high probability and to one branch of R-tree as a result.

Additionally to spatial join queries, R-trees allow to efficiently evaluate k-NN queries, as well as obtaining nearest neighbors iteratively [43, 44]. This feature is exploited by SIA as a part of the pruning technique, that reduces necessary memory requirements and accelerates edge sorting for each particular node. We will use this feature as well in our novel approach for

general graphs. Recently a novel algorithm was introduced that uses highly-efficient concurrent priority queues in order to evaluate Breadth-First k-NN queries in parallel [45]. This algorithm can be embedded in both our algorithm and SIA as future work.

1.5 Cost-Scaling algorithm

Cost-Scaling Algorithm (CSA) is an exact algorithm for Circulation Problem. Originally, it was proposed by Goldberg and Tarjan in 1990 [17], recently new time bounds and implementation details were presented in [18]. The algorithm also belongs to Hungarian-based (see section "Exact algorithms"), as it also exploits principles of inverting edges, node potentials and successive shortest path searches. In fact, the core of the algorithm is SSPA, that is a successive subproblem in CSA which is solved with iteratively inreasing precision.

In order to describe the algorithm in details, we should mention some more definitions. A pseudoflow is a flow that does not satisfy conservation constraints, i.e. the sum of input and output flows may not be equal. If an input flow in a node is prevailing, then we call the node an excess. If an output flow is greater, then the node is a deficit. c_p is a reduced cost of an edge.

$$c_p(v,w) = c(v,w) + p(w) - p(v)$$

$l(v,w) = \lfloor \frac{c_p(v,w)}{\epsilon} \rfloor + 1$ is a distance function that defines a distance between two adjacent vertices for shortest-path search.

A circulation f is ϵ-optimal with respect to a price function p if for every arc (v,w), we have

$$c_p(v,w) < -\epsilon \Rightarrow f(v,w) = u(v,w) \tag{1.5}$$

The pseudocode is presenten in Algorithm 4. It performs in iterations. One iteration is an iteration of a *while* loop in $CalculateMaxFlow$ function. In each iteration there are 3 phases:

1) Decrease ϵ by 2
2) $raisePotentials$

3) Compute blocking flow in G_A

Last two phases are repeated until f becomes a circulation. At the end of each iteration a circulation is ϵ-optimal. This is the main invariant of the algorithm. ϵ value decreases from the maximum edge cost value to $\frac{1}{|V|}$. Since we consider integer capacities, ϵ-optimality with $\epsilon = \frac{1}{|V|}$ is equivalent to the total optimality [17].

Algorithm 4 Cost-Scaling Algorithm (CSA)

```
1:  function CalculateMaxFlow(G,{s,t})
2:      ε = C, p = 0, f = 0                          ▷ C - global maximum cost
3:      while ε ≥ 1/n do
4:          (ε,f,p) = refine(ε,f,p)
5:      return f
6:  function Refine(ε',f',p')
7:      ε = ε'/2, f = f', p = p'
8:      for all (v,w) ∈ E_A do
9:          f(v,w) = u(v,w)
10:     while f is not a circulation do
11:         RaisePotentials(ε,f,p)
12:         f = f+ a blocking flow in G_A
13:     return (ε,f,p)
14: function RaisePotentials(ε,f,p)
15:     add new vertex r
16:     for all excesses v do
17:         add arc (r,v) with d(r,v) = 0
18:     Initialize Dial's SSSP algorithm with root in r
19:     repeat
20:         Make Dial's SSSP algorithm iteration w.r.t l
21:     until first deficit u is found
22:     for all v : d_min(r,v) < d_min(r,u) do
23:         p(v) = p(v) + (d_min(u) − d_min(v))ε
```

The correctness of CSA is based on maintaining the ϵ-optimality of the flow. This property we describe in Lemma 3. CSA has some improtant properties. By linear programming duality, 0-optimal circulation is optimal. A circulation is optimal if and only if there exists a potential function such that all residual arcs have non-negative reduced costs. A circulation is optimal if and only if the recidual graph has no cycles of negative cost. [17, 18] A more detailed analysis of properties of the algorithm, as well as analysis of similar algorithms, like Negative Cycle Cancelling and Capacity Scaling algorithms, are available in [46].

Lemma 3. *ϵ-opitimality of a circulation f guarantees the optimality of f in CSA*

Proof. The lemma is equivalent to the Theorem 2.3 in [17]. Consider a simple cycle in G_f. The ϵ-optimality of f implies that the reduced cost of the cycle is at least $-n\epsilon > -1$. The reduced cost of the cycle equals its original cost, which must be integral and hence nonnegative. So, there are no negative cycles in a recidual graph, so a circulation is optimal. □

1.5.1 Intuition under ϵ parameter

ϵ parameter in CSA is the most significant specialty that differs this algorithm with SIA or any other algorithm that uses graph traversal, inverting edges and potential function. Here are some suggestions and notes about what intuition can be applied towards this parameter based on the observations and features of CSA.

First of all, this parameter is a parameter for coarse-graining the flow. After each iteration ϵ-optimality holds, that means a flow saturates each arc where reduced cost $c_p(v,u) = c + p(v) - p(u) > -\epsilon$. So, we say that flow is ϵ-optimal. It can differ from the optimal one not more than of ϵ.

Another role that this parameter has is the speed of change of potential function. The potential throughout the algorithm "suggests"the flow where is the direction to deficits. When traversing a graph, the algorithm tries to go to the direction that is not yet "discovered that contains nodes

farthest from the excess nodes. This is archived by increasing the cost of going in the direction towards the excess.

One more very important observation is that ϵ parameter shows a balance between weight and hop priorities in the graph traversal, where a hop is one "step"between two nodes along one edge. In other words, the number of hops is an alternative distance cost measure where a cost of a path between two nodes is equal a number of edges between two nodes along that path. For example, for a linear graph with 3 nodes sequentially connected $A \to B \to C$, where a weight of the first edge is 2 and the second one is 3, the cost of the total path is 5 and there are two hops in that path.

The reason why ϵ shows the balance is that this parameter is added to the each reduced cost of any edge during Dijkstra shortest path search. So, if the parameter is small, Dijkstra goes as usual and searches for the shortest path according to the weights of edges. But if ϵ is large, then at each iteration Dijkstra prefers to reduce the number of hops, because each hop increments total cost of the path at least on ϵ that is more than most of the weights in the graph. For example, suppose there are two possible paths from source to target in a graph with one edge and two edges respectively. The first path contains one edge with a weight that is greater that the sum of two weight in the second paths. If ϵ is small, the first path is preferable because weights are the true argument. In the opposite case, the second path is preferable, because the total cost of the second path is at least 2ϵ that is greater than $\epsilon + w$ with any weight w, since $\epsilon > w$ with high probability.

1.5.2 Variations and Improvements of CSA

Many variations of CSA have been proposed in the literature. The same principle of increasing precision was presented in original paper [17], where capacity scaling is used instead of cost scaling. Both capacity and cost scaling also can be combined [47].

In the latest work of Goldberg et al. [18], authors present a novel time upper bound for bipartite minimum matching problem that is equal to $O(\sqrt{r}m \log C)$. Since the same framework is used as in maxflow problem case, this lead to the time bound of $O(\sqrt{r}D \log C)$ rounds, where $r = \min |V_1|,|V_2|$.

1.6 Depth-first Cost-Scaling algorithm

Lemon library [48] provides a very efficient implementation of CSA. The solution includes a slightly different approach for raising potentials and contains several heuristics that are described below.

Potential values are changed during DFS traversal of a graph. This is a major difference between current implementation and the original presented by Goldberg. If a node does not have adjacent admissible edges, then it is relabeled according to the following formula:

We define $E_f = \{(v,w) \in E | f(v,w) < u(v,w)\}$ as a set of residual arcs and $e_f(v) = \sum_{(w,v)\in E} f(w,v)$. A vertex v is an excess if $e_f > 0$ and a deficit if $e_f < 0$.

$E_A = \{(v,w) \in E_f | c_p(v,w) < 0\}$ is defined as a set of admissible arcs and G_A is admissible subgraph with edges E_A, accordingly.

A function $f : E \to R$ is a pseudoflow if it satisfies two conditions:

1. $f(v,w) = -f(w,v)$
2. $f(v,w) \leq u(v,w)$, where $u(v,w)$ is capacity constraint.

$$p \leftarrow p - (\min c_p + \epsilon)$$

where $c_p = \pm weight + p[target] - p[source]$ and minimum value is taken over all adjacent non-saturated edges.

An intuition behind this formula is the following. Since we are considering all outgoing edges (including inverted edges), $p = p[source]$. So, we get

$$p = p[s] = p[s] - weight + p[t] - p[s] - \epsilon = p[t] - weight - \epsilon$$

$$c_p = weight - p[t] + p[s] = weight - p[t] + p[t] - weight - \epsilon = -\epsilon$$

As a result, we get a new admissible arc for the arc with a minimum cost, such that ϵ-optimality in the graph is preserved. Furthermore, we are trying to maximize the number of new admissible arcs for the processed node when changing the potential. This is done by choosing maximal possible decrease of the potential.

The formula is actually the same for latest Goldberg, because $l = c_p/\epsilon + 1$ is $l = c_p + \epsilon$, but multiplied by ϵ. That means maximum distance in Dijkstra now is not $3n$, but $3n * \epsilon$.

Because of this behavior, we'll use the notation of Depth-First Cost-Scaling Algorithm (DF-CSA) referring to the implementation of CSA by authors of Lemon library.

1.6.1 Implementation details

In this section, we will describe the implementation in details. The code is presented in Algorithm 5. There are several significant improvements over the simple cost-scaling, that are used in the library.

GOE supply type Greater or equal supply type is supported by the algorithm. The realization of this feature is done by adding one additional "root"node, that have an arc to each other node. The costs of such arcs are zero, but the capacities are set separately for direct and inverse arcs. For direct arc from a node to the root node is equal to an excess value obtained after calculation of a feasible flow. The sum of such excesses is equal to the total sum of supply in the graph, i.e. all excess supply provided by the problem. For an inverse arc to the same node from the root, the capacity is equal to the sum of supplies plus 1.

Internal Method Three internal methods are implemented: Push, Augment, and Partial Augment. The last one is stated as default one and the most efficient. These are methods how the base operation is performed. In Push method, a flow is moved only on one admissible arc at once. In the augmenting method, a flow is moved on admissible paths from a node with excess to a node with the deficit. Partial augment is a combination, when a flow is moved on admissible arcs from excess to a deficit, but only up to some maximum length.

Heuristics There are two implemented heuristics: priceRefinement and globalUpdate. These heuristics improve the potential function throughout the algorithm. Detailed description available in [20]. One

heuristic, called *speculative arc fixing*, is suggested by authors as a future work.

Potential refinement heuristics introduces an additional step at the beginning of each phase to check if the current solution is already (ϵ/α)-optimal. This step attempts to adjust the potentials to satisfy the (ϵ/α)-optimality conditions, but without modifying the flow. If it succeeds, the refining procedure is skipped and the next phase begins. Results of [20] verified that this heuristic substantially improves the overall performance of the algorithm in most cases.

Global Update heuristics is called if too much relabels during one iteration appeared. The basis for this heuristic is the observation that if there are two subsets of nodes S and $\bar{S}$, there are no admissible arcs between them, all deficit nodes are in S and at least one active node is in $V \backslash S$, then the potential of nodes in S can be increased by ϵ without violating ϵ-optimality.

Algorithm 5 DF-CSA

1: **function** INITIALIZATION
2: $p \leftarrow 0, excesses \leftarrow supply$
3: Saturate all arcs with $cost < 0$
4: $\epsilon = \max_i cost(i) \cdot n$
5: Find feasible flow by Push-Relabel algorithm
6: Check feasibility by running Push-Relabel
7: **function** COST-SCALING(Graph)
8: Initialization
9: Push or Augment method
10: **if** Solution is not optimal **then**
11: Run Bellman-Ford and update potentials
12: Shift potentials
13: Handle non-zero lower bounds
14: **function** AUGMENT(Maximum Distance)
15: path = FindPath(Maximum Distance)
16: AugmentPath(path)
17: **function** AUGMENTPATH(Path)
18: **for all** nodes in the path **do**
19: Increase flow on $\min\{resCap, excess\}$ ▷ Maintain only residual capacities and excesses, not flows
20: Push to Ω if there is an excess

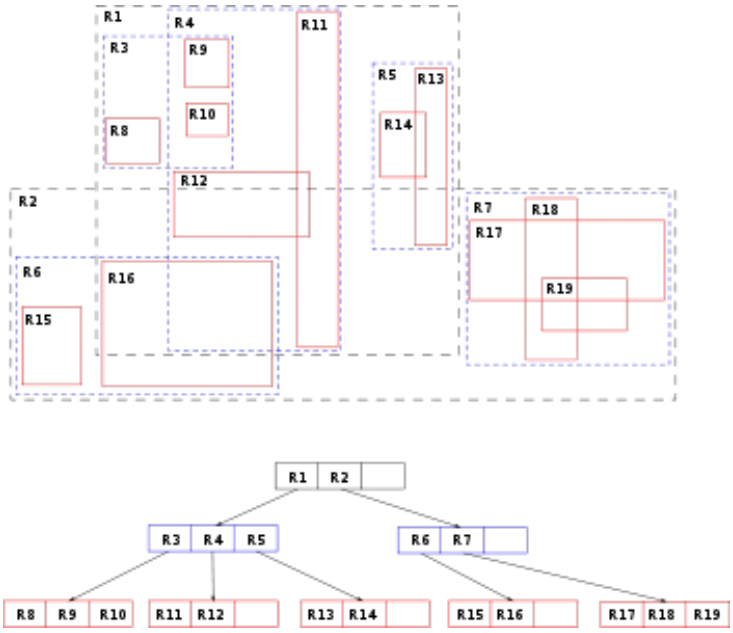


Figure 1.1 — A sample of R-tree structure

Algorithm 6 DF-CSA FindPath function

```
1:  function FindPath(Maximum Distance)
2:      Saturate arcs with negative c_p
3:      Active nodes Ω ← excesses
4:      while There are active nodes do
5:          Pop nodes with negative excess from Ω
6:          v ← pop from Ω
7:          while Path is less than Maximum Distance and v is Excess do
8:              for all Out arc e of v do
9:                  if e is not saturated then
10:                     Current node ← next neighbor : c_p < 0
11:                     if Cycle is found then return Cycle
12:             Calculate min c_p for edges of current node
13:             p[current node]− = min c_p + ϵ
14:             Current node ← path.previous
15:         Global Update Heuristics
```

2 Design

2.1 Graph representation

Internal graph representation in all algorithms is based on adjacency list principle, used in such libraries as Lemon library and Ligra library [49]. It implies storing a list of ids of target nodes of each edge Ngb. Edges in Ngb are sorted by a source node, i.e. for each source node, all outgoing edges are stored near each other in Ngb. Another list $FirstOut$ indicates where a set of outgoing edges for a particular node starts. This approach is highly efficient, as it exploits such low-level acceleration techniques as processor caching and branch prediction. Separate arrays store values of costs, an id of correspondent inverted edge (or direct edge, accordingly), a list of ids of source and target nodes for each edge, a boolean array that states if an edge in inverted, and finally an array that stored an available capacity. Another array stores the current excess values for each node. Note, that this set of arrays allows skipping explicit storage of current flow, an upper bound of a flow and supply values of nodes. However, this approach does not allow to dynamically add new edges, so instead of continuous array Ngb and a list $FirstOut$, a two-dimensional array is used, where the first dimension is an id of a node, and the second dimension is a dynamic list of neighbors, each representing an outgoing edge.

In this representation, inverted edges are utilized the same way as direct ones. One approach is a two-dimensional array of output edges that contains only those edges which have a non-zero available flow, we found that the approach of monotonic increase of the array is significantly faster for the used case of unit capacities. This is due to the "alternating-path"behavior of and hence the necessity of removing empty edges after each flow increase and adding inverted edge to a neighbor. For S-DF-CSA, we do not take into consideration adding edges as well, in order to compare with original DF-CSA. We retained the continuous versions of an adjacency list Ngb and $FirstOut$ and implemented edge addition by a new array $LastOut$ that stores the last edge that was "activated"so far for each node.

2.2 Modified SIA

In this section, we derive a possible improvement over SIA algorithm. Firstly, we would like to note, that, according to the definitions provided in [4], in this section we consider PUA (Path Update Algorithm), that is a modification over SIA, that allows successively update results of Dijkstra shortest path computation while adding new edge in E_{sub} (see Section 1 for details), however the basic idea remains the same. Our experiments confirm that SIA without PUA is significantly slower and does not have enough benefit from incremental edge addition since the algorithm has to rerun Dijkstra each time a new edge is added and a cost of this operation surpasses a benefit from pruning.

In the case of spatial data, it is important to note, that shortest path computation as a subroutine in algorithms that use the notation of inverted edges, has a slightly different meaning comparing to a single shortest path computation in a general graph. When computing shortest path in a graph with inverted edges, some hops may go trough inverted edges and, as a result, a flow through inverted edge may be increased. Such operation, as was mentioned before, is equivalent to canceling of a certain amount of flow along a direct edge, that corresponds to the inverted one.

Especially this is important to remember in case of bipartite matching, because while running an algorithm, the shortest path can contain hundreds of edges while the total cost will not be greater than an average edge weight. This is one of the reasons why pruning proposed in SIA has a very high rate of pruned edges. For bipartite case, however, a total length of shortest paths are anyway quite small, as will be illustrated in Section 3.

In spite of the small total length of shortest paths and unitary length domination for a bipartite case, Dijkstra calculation remains a bottleneck for the algorithm. If a node hasn't been assigned correctly, reassigning can lead to dozens of hops in the shortest path and rapid increase of a potential value of many visited nodes. In SIA, a threshold that guarantees a feasibility of last Dijkstra execution is the following:

$$v_{min}.\alpha > TopKey(H) - \tau'_{max}$$

In this equation, according to definitions in the paper, v_{min} is a minimum distance to a target (node with negative supply, if using network flow definitions), $TopKey(H)$ is a top key of the global heap and τ'_{max} is a maximal value among potentials of nodes that belong to the first subset of bipartite graph. Global heap stores a sum of shortest distance to each of visited nodes and corresponding potential:

$$globalH \leftarrow \min_{visited} v.\alpha + p(v)$$

Increasing a length of the shortest path and a great number of nodes with updated potentials may lead to a perceptible influence of maintaining τ_{max} value on a performance. If a non-complete bipartite graph is used, or there are any additional restrictions on connections, or a general graph is used instead of bipartite (as we will show further), the problem may become even worse.

A new value in the heap that does not need any potential maintaining is presented in the equation 2.1. It allows skipping any potential maintaining since it uses only the potential value of a node that we currently are considering (a node that is being updated or a newly visited node). The Lemma 4, based on the Theorem 3.7 in [4], proves the optimality of a matching in SIA with the new threshold. We will use notations presented in this thesis.

$$globalH \leftarrow weight - p(v) \tag{2.1}$$

Lemma 4. *Consider an edge set $E_{sub} \subseteq E$ and a shortest path in E_{sub} from a source to a target, where for each node v that was visited during shortest path computation, the equation 2.2 holds.*

$$\forall (v,w) \in E \setminus E_{sub} \rightarrow c(v,w) - p(v) > \Theta \tag{2.2}$$

If a total cost of the shortest path is smaller than Θ, then the shortest path in E_{sub} is also the shortest path in E.

Proof. Suppose there is a shortest path A from a source to a target in E_{sub}, such that the equation 2.2 holds for each node in the path. Let there be another path B that is shorter, but includes at least one edge e from the set

$E_{sub} \setminus E$. Such path must contain at least one edge of type $(v,w) \in E_{sub} \setminus E$, where v is a visited node during shortest path computation. Otherwise, there can be no connection (continuous path) from the source node to the edge e, since the source node is a visited node. Let c_p be a reduced cost of the edge (v,w). The total cost of B must be at least c_p large, because $c_p > 0$ (flow feasibility property of the algorithm). At the same time, because of non-negativity of potentials, the following holds:

$$c_p = c(v,w) + p(w) - p(v) \geq c(v,w) - p(v) > \Theta$$

So, the total cost of B is at least as large as the reduced cost of one edge in B, that is larger than the total cost of A. □

Despite the fact, that the new threshold condition is weaker than the original one, it exploits the benefit of ignoring any potential values except a potential value of the enheaped edge itself. This could lead to a better performance for some types of graphs.

2.3 Spatial-Optimized Cost Scaling Algorithm

In this section, we present a new algorithm that combines CSA and a pruning technique of SIA. We call it Spatial-optimized Cost Scaling Algorithm (S-CSA), since it exploits a possibility of incremental nearest neighbor search, as SIA does. In order to describe the algorithm, we prove several lemmas first. We use the CSA notation of the reduced cost of an edge $c_p(v,w) = c(v,w)+p(w)-p(v)$ and the length function $l(v,w) = c_p(v,w)+\epsilon$. The contribution of one edge in the shortest path search is equal to the length function value on that edge. In each lemma we considered a graph $G(V,E)$ with distance-bounded edge subset $E_{sub} \subseteq E$.

One of the differences between CSA and S-CSA is that all costs in S-CSA are multiplied by $\alpha \cdot \epsilon$. Distance function l is changed accordingly: instead of $l = \lfloor \frac{c_p}{\epsilon} \rfloor + 1$, in S-CSA $l = c_p + \epsilon$ is used. The same modifications are applied to the pruning threshold and minimum ϵ value requirement (1 instead of $\frac{1}{|V|}$). This allows eliminating float number operations while calculating flows throughout the algorithm.

Lemma 5. *Let f be an ϵ-optimal pseudoflow in E_{sub}. If eq. 2.3 holds, then f is also ϵ-optimal in E.*

$$\forall v \in V \rightarrow \min_{w \in E \setminus E_{sub}} c(v,w) - p(v) + \epsilon > 0 \tag{2.3}$$

Proof. Since f is ϵ-optimal in E_{sub}, then

$$\forall (v,w) \in E_{sub} \rightarrow c_p(v,w) < -\epsilon \Rightarrow f(v,w) = u(v,w)$$

where $u(v,w)$ is a capacity of the edge. Eq. 2.3 imply that

$$\forall (v,w) \in E \setminus E_{sub} \rightarrow \min_{w \in E \setminus E_{sub}} c(v,w) - p(v) + \epsilon > 0$$

Because of the positiveness of p, we have

$$c_p(v,w) > c(v,w) - p(v) > -\epsilon$$

So, if a reduced cost is too small, then an edge is guaranteed to be in E_{sub}, where the ϵ-optimality is true.

$$\forall (v,w) \in E \rightarrow c_p(v,w) < -\epsilon \Rightarrow (v,w) \in E_{sub}$$

□

Lemma 6. *Let sp be the shortest path between an excess s and a deficit t in E_{sub} according to the length function $l : E \rightarrow \mathbb{N} \cup \{0\}$. Let $mindist(v)$ be a shortest path length from s to v and Θ be defined by eq. 2.4. If eq. 2.5 holds, then sp is also the shortest path in E.*

$$\Theta = \min_{v \in sp} \{mindist(v) + \min_{w \in E \setminus E_{sub}} c(v,w)\} \tag{2.4}$$

$$mindist(t) \leq \Theta - \max_{v \in sp \,:\, mindist(v) < \Theta} p(v) + \epsilon \tag{2.5}$$

Proof. Suppose there is a path from s to t that contains at least one edge (u',w') from $E \setminus E_{sub}$ with a total length smaller than length of sp. Since every path is continuous and has at least one common node s, then there should be at least one node $u \in sp : (u,w) \in E \setminus E_{sub}$. Each path throught (u,w) is at least as long as $mindist(u) + l(u,w)$, because the total length of

the path is a sum of l function values and $l \geqslant 0$ for each edge in the graph because of the ϵ-optimality invariant. By definition,

$$l(u,w) = c(u,w) + p(w) - p(u) + \epsilon$$

Let $mindist(t)$ be a sp length and $mindist'(t)$ be a shortest path length that goes throught (u,w). Given eq. 2.5 and taking into consideration non-negativity of p and c functions, we have

$$mindist(t) \leqslant \Theta - \max_{v \in sp\,:\, mindist(v) < \Theta} p(v) + \epsilon \leqslant$$

$$mindist(u) + c(u,w) - \max_{v \in sp\,:\, mindist(v) < \Theta} p(v) + \epsilon \leqslant$$

$$mindist(u) + c(u,w) - p(w) + \epsilon \leqslant$$

$$mindist(u) + c(u,w) - p(w) + \epsilon \leqslant$$

$$mindist(u) + c(u,w) + p(u) - p(w) + \epsilon \leqslant mindist'(t)$$

Contradiction. □

The complete pseudocode of S-CSA is presented in Algorithm 7. W.l.g., we describe an algorithm in terms of spatial data as an input. Spatial data is described by a set of nodes in space with some of them marked as source and target nodes. Spatial index (R-tree R) is used in order to obtain iteratively nearest neighbor for each node. This operation is denoted as $R.NN(v) \to E$ for a node v. $R.NN(v)$ can be thought as an abstract function that returns next smallest edge of a node. It can be done by any other index, or, in the case of general graphs, by iteration through sorted edges for each node. In the pseudocode, we omit a special case when there are no more edges to add for a particular node, but there is such possibility, so this must be covered in the full version of the code. Additionally to that, we use the following conventions:

— $C = \max_{e \in E} c(e)$

— A source node is an excess which is used as a start node in a Dijkstra execution

— V_e is a subset of vertices that are excesses (input flow is greater than output).

— V_d is a subset of vertices that are deficits

— $mindist(v)$ is a minimum distance from a source node in the current Dijkstra Execution, according to the length function $l(v,w) = c_p(v,w) + \epsilon$.

— α is a parameter that states the pace of decreasing of ϵ at each iteration. In this work, we use $\alpha = 16$.

— In order to simplify notations we assume that for each edge (v,w), (w,v) is an inverted edge, i.e. for each node pair there is only one-direction direct edge.

Heap-based Dijkstra algorithm is used for the shortest path search from a source node to any deficit. It retrieves as an input a source node s, a subset of edges E_{sub} and a heap H_d and array *mindist* which it uses as an initial setup. H_d is used in order to reuse results from the previous unsuccessful attempt to find the shortest distance, that failed because of the threshold violation. Dijkstra returns a list of edges in a shortest path from s to a target deficit t, a target node t, a new resulting heap H_d and *mindist* array. t can be None if there is no path from a source to a deficit. The correctness of such approach is proved in Lemma 7. We call H_d a correct heap if Dijkstra execution based on H_d leads to correct *mindist* values. s node is fixed. Lemma 7 also shows that PUA algorithm (see Section about SIA), that uses the additional heap for updating Dijkstra heap, is not necessary.

For example, consider 4 nodes $\{A,B,C,D\}$, where A is a excess and D is a deficit. The topology is depicted on Figure 2.1. Dashed edge is an edge which is not yet discovered. Suppose Dijkstra is executed on the graph. First, A is enheaped. Then, A is deheaped, B and C enheaped. At this moment the heap contains 2 nodes - B and C, and *mindist* array contains values: A:0, B:1, C: 10, D: ∞. The heap together with mentioned *mindist* values are obviously correct, that means if Dijkstra starts with such values, it will obviously finish with the same result as with empty initial heap. Now suppose Dijkstra runs until the end. Final *mindist* looks like A:0, B:1, C:10, D: 21, and the final heap is empty. Now, we add new edge - BC and update *mindist* according to S-CSA algorithm. New $mindist(C) = 3$ and C is enheaped. This heap and array state are also correct, because if Dijkstra continues execution, then D

will be updated in one step, and distance to D and C is the same as it would be if Dijkstra runs starting from A.

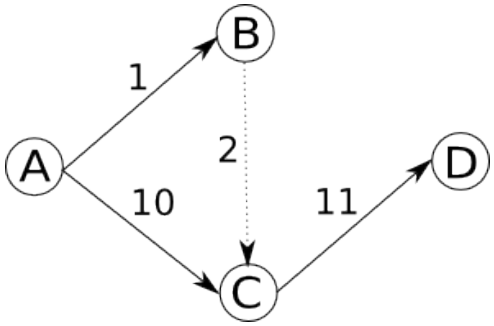


Figure 2.1 — Example of Dijsktra Execution

The Algorithm 7 goes in iterations, each has several steps. For each ϵ value (line 3), we first saturate all edges with negative reduced cost (lines 6-10), then a flow is increased until there are no excesses left (lines 11-29). In order to preserve ϵ-optimality after the flow increase, we first run an SSSP algorithm (lines 12-25) and increase potentials for those nodes (lines 26-27), where a distance from an excess is shorter than a distance to the nearest deficit. Then, we increase a flow along the path from an excess to a deficit (lines 26-27). SSSP algorithm runs on a subset of edges E_{sub}. New edges are added to E_{sub} (lines 16-18) until the shortest path found is guaranteed to be the shortest path in E (line 15). Theorem 1 proves the correctness of the algorithm.

Lemma 7. *If H_d is a correct Dijkstra heap in E_{sub} for a source node s and an array of mindist values, then H_d is also a correct Dijkstra heap for $E_{sub} \cup (v,w)$ if*

1) $mindist(w) \leftarrow \min\{mindist(w), mindist(v) + l(v,w)\}$
2) w is in H_d if mindist becomes updated

Proof. If w is not enheaped and *mindist* is not updated, the correctness is trivial. Suppose, we have a correct H_d and w is enheaped. This preserves the correctness of Dijkstra algorithm. There are two possibilities. First, if a new *mindist* is greater than current top value of H_d. Then, (1) and (2) together are equivalent to one iteration in Dijkstra algorithm. If it is smaller, then the next iteration of Dijkstra algorithm will start from v and as a result all

mindist values that should be influenced by a new edge (i.e. all nodes that already have been visited by Dijkstra but after adding new edge a shorter distance has become available) eventually will be updated. So, H_d is still correct since it will eventually lead to a correct *mindist*. □

Theorem 1. *Resulting flow in S-CSA is an optimal solution of a Circulation Problem.*

Proof. We show that ϵ-optimality invariant is preserved in our algorithm, the same as in CSA algorithm. Lemma 3 proves that holding the invariant guarantees the optimality.

An admissible edge is an edge with a negative reduced cost is negative. If a flow is changed only for admissible edges, then ϵ-optimality preserves because, after saturating admissible edge an inverted edge becomes non-full, but if a direct edge has negative reduced cost, then it is positive for an inverted edge according to the formula of a reduced cost. The same for an admissible inverted edge.

We prove the invariant by induction. At the beginning $p(V) = 0$, so the invariant holds. Suppose at the beginning of the iteration ϵ-invariant holds. After decreasing ϵ by α, some edges may violate ϵ-optimality. By saturation of such edges, a flow becomes again ϵ-optimal for a smaller ϵ. Lemma 5 guarantees that all such edges in E become saturated. After saturation, a flow may become a pseudoflow. While there are excesses (a pseudoflow is not a flow), we push a flow from excesses to deficits along admissible paths. Invariant holds during raising flow as was mentioned above. If there are no admissible paths, potentials are increased by finding SSSP from an excess to a deficit. Lemma 6 proves the correctness of SSSP step.

Potential is increased for each visited in SSSP step node on a value of $mindist(t) - mindist(v)$. We refer to the Lemma 6 in [18] that proves that this change of potential values is equivalent to the following: while there is no path from an excess to a deficit - increase by ϵ potential of all nodes that are reachable from s by admissible edges. Furthermore, there is a proof that any *mindist* value is either infinity (a node was not reached by SSSP) or is

Algorithm 7 S-CSA

Input: V, R
$E_{sub} = \emptyset$, $\epsilon = C$, $p(V) = 0$, $p_{max} = 0$
while $\epsilon > 1$ or $|V_e| > 0$ **do**
    $\epsilon \leftarrow \epsilon/\alpha$
    for all $v \in V$ **do**
        while $c(R.NN(v)) - p(v) + \epsilon < 0$ **do**
            $E_{sub} \leftarrow R.NN(v)$
        for all $w : (v,w) \in E_{sub}$ **do**
            $f(v,w) \leftarrow u(v,w)$ ▷ Saturate the edge
            $f(w,v) \leftarrow 0$ ▷ Free the inverted edge
    while $|V_e| \neq \emptyset$ **do**
        $H, H_d \leftarrow$ empty min-heaps
        $s \leftarrow$ any node from V_e
        $H \leftarrow c(R.NN(start))$
        while $mindist(t) > H.TopValue - p_{max} + \epsilon$ or $t =$ None **do**
            $(v,w) \leftarrow R.NN(H.TopKey)$
            Deheap (v,w) from H
            $E_{sub} \leftarrow (v,w)$
            if $mindist(w) > mindist(v) + l(v,w)$ **then**
                $mindist(w) \leftarrow mindist(v) + l(v,w)$
                $H_d \leftarrow w$
            $sp, t, H_d, mindist \leftarrow$ DIJKSTRA($R.NN$, E_{sub}, H_d, $mindist$)
            for all visited v **do**
                $H \leftarrow mindist(v) + c(R.NN)$ ▷ Update the heap
            $p_{max} \leftarrow \max_{v \in V : mindist(v) < H.TopVal + \epsilon} p(v)$
        for all visited v **do**
            $p(v) \leftarrow mindist(t) - mindist(v)$ ▷ Update the heap
        for all $e \in sp$ **do** ▷ Push maximum available flow from s to t
            $f(e) \leftarrow \max_{e' \in sp}\{u(e) - f(e)\}$

less than $3n\epsilon$. We do not exploit this property because we use a heap-based Dijkstra.

Increasing a potential of each reachable node by ϵ does not break the invariant. Suppose we have two sets of nodes S and $\bar{S}$, where S are those which are reachable by accessible edges and $\bar{S}$ which are not. If we increase by ϵ all potentials in S, then reduced costs of all edges that have adjacent nodes both in S or both in $\bar{S}$ do not change, since potentials of both nodes are changed on the same value. Consider an edge (v,w), which partially lie in one set, partially in another (it can be both $v \in S$ or $v \in \bar{S}$): since one adjacent node is not in S, then $c_p > 0$. When a potential is changed, c_p can change not more than on ϵ, so $c_p > -\epsilon$ after raising potentials, that is ϵ-optimal value. This proves the invariant after increasing potentials in S-CSA and finishes a proof of holding the invariant thought the algorithm. □

2.3.1 Error parameter dynamics

As in CSA, ϵ can be thought as an error parameter, that allows a circulation be approximately optimal. In S-CSA we use decreasing of the parameter the same way as in original CSA, but with the denominator of $\alpha = 16$. However, in the proof of Theorem 1, that proves the correctness of the algorithm, the only obligatory property of ϵ for the correctness of the algorithm is that at the end of execution ϵ must be small enough (smaller or equal to 1) to guarantee the optimality of the circulation. At each iteration of the algorithm where we scale ϵ, there is no any condition on how it is scaled. The change in ϵ defines the number of edges that violates ϵ-optimality and hence must be saturated. On the one hand, this number should be minimized to increase performance. On the other hand, slowing down this step, we might increase the performance of Dijkstra execution by reducing a number of hops needed for reaching a target.

A discrete function $\epsilon = \epsilon(T_i)$, where T_i is an iteration number, is a possible parameter of the algorithm that could dramatically influence the performance. The only condition on the function is $\lim_{i\to\infty} \epsilon(T_i) = 1$. It can even violate monotonicity. For example, if during the execution a lot of very large edges have been added to E_{sub}, then it is likely that increasing ϵ

will bring more benefit than decreasing or remaining the same. Note, that an extreme case of the function is $\epsilon(T_i) = 1$. This makes S-CSA a simple generalization of SIA on a general graph and cancels any cost scaling. We compare this case with S-CSA described in Algorithm 7.

2.4 Pruning in DF-CSA

As was mentioned in Section 1, DF-CSA is one of the most efficient implementations of CSA. We also suggest a simple pruning technique for DF-CSA, called Spatial-optimized Depth-First Cost Scaling Algorithm (S-DF-CSA). The technique is based on the assumption that each node either has all outgoing edges sorted or can obtain them incrementally in the heap manner, similarly to S-CSA.

The algorithm is the same as DF-CSA (Algorithm 5), but we only consider some subset of edges E_{sub} and after each for loop at line 25, for a current node v we add next smallest outgoing edges from $E \setminus E_{sub}$ to E_{sub} that satisfy the following equation:

$$c(R.NN(v)) - p(v) \leqslant \min_{u:(v,u)\in E_{sub}} c_p(v,u) \tag{2.6}$$

Here we use the same notation $R.NN$ as in previous section. Lemma 8 proves the correctness of S-DF-CSA.

Lemma 8. *If distance to the next nearest neighbor of a node v satisfies the equation 2.7, then the current minimum reduced cost is also minimum among all outgoing edges of a node in the full graph. Hence, the value which is added to the potential of the node based of E_{sub} is the same as the value based on E, which preserves ϵ-optimality.*

$$c(R.NN(v)) - p(v) > \min_{u:(v,u)\in E_{sub}} c_p(v,u) \tag{2.7}$$

Proof. Equation 2.7 is equivalent to

$$c(R.NN(v)) - p(v) > c_{min} + p(t) - p(v)$$

Simplifying, we get

$$c(R.NN(v)) > c_{min} + p(t)$$

For each outgoing edge e of the node v the following holds

$$\forall e \to c(e) + p(t') > c > c(R.NN(v)) > c_{min} + p(t)$$

As a result,

$$\forall e \to c(e) + p(t') - p(v) > c_{min} + p(t) - p(v)$$

Last equation formulates the statement of the lemma: current reduced cost is the minimum over all outgoing edges. □

2.5 Distributed CSA

Although the original CSA was introduces as parallelizable and with a possible distributed implementation [17], the novel algorithm analysis [18] does not explicitly describe distributed version. We derive a list of possible improvements and important notes about the distributed version of lately published CSA, as well as present its complexity analysis.

A fundamental question in the area of distributed graph theory that has been actively studied for many years is how much time complexity is needed to solve a problem in CONGEST model. [21], [50], [51]. The notable property of current algorithm is that is only unit massages are passed through each channel per iteration.

Each process manages its own value of p_i and can contain the flow value f_i for outgoing edges. It takes $\lceil \log_2(n) \rceil$ iterations until $\epsilon < \frac{1}{n}$ to termination. Further, we describe every part of CSA algorithm separately.

2.5.1 Raising potentials

In the paper, sequential *raisePotential* is implemented by creating one additional vertex r, that is the source, and solving SSSP problem with multiple targets, that are deficits. In distributed version r is not needed. A process at the beginning of *raisePotentials* phase checks if it is an excess and start its own SSSP search using synchronous BFS, that takes $O(m)$ time. Two searches that started at different nodes may interfere, so that if a node x was reached by one search and a distance d was set to x as a shortest distance to the origin of the first search, and later another search claims that

the shortest distance to the second source is smaller than d, then d is set to the smallest distance.

The same argument about iterations as in section 2.5.2 can be applied here. We define the end of *raisePotentials* phase as a moment when at the end of SSSP algorithm iteration one of the runs started in one of the processes reach a deficit node. At that moment, every other node in any connected component has either infinite distance to the source of SSSP or a distance that is not greater than the distance to u that was found in the current iteration. So, if the global end of current phase was announced, then each process can decide about the value Δp_i. If a node has infinite distance, then a p_i value is not updated. If a node changed its shortest distance from infinity to some value exactly at the last iteration before phase transition, then it also does not update p. Otherwise, it should update p using its d_l value and $d_l(u)$ value that could be broadcasted by synchronizer.

Authors describe a faster implementation of *raisePotentials* using Dial's shortest path algorithm. However, as was shown in the section about SSSP, this is not the most efficient approach in a parallel setting. For example, Delta-stepping algorithm can be used instead. Dial's algorithm allow early termination, so it may be faster, but this benefit is advantageous only in sequential mode.

2.5.2 End of Iteration

Another implementation detail that should be mentioned is the end of an iteration. The algorithm repeats phases until a flow becomes a circulation. In distributed setting, the end of the iteration is defined by a synchronization method, that can be a synchronizer. After a blocking flow calculation phase, each process knows if it is an excess. Let *raisePotentials* phase go after each blocking flow calculation phase. At the end of *raisePotetials* a synchronizer should know that every process has finished current phase. But a process can additionally notify the synchronizer if the process actually did at least something at the current phase, i.e. if it has been in an excess subset of vertices. A synchronizer then can decide, if nobody was active in this phase, then start new iteration, otherwise start new phase again. Although

we consider a synchronous model of distributed system, this way using a synchronizer is legitimate, since it equivalent to the definition of the end of an iteration in a certain way, which includes a condition to the inner state of each process at the end of a phase. Note, that this also implies that message complexity should not include this synchronization costs.

However, this makes our algorithm centralized, since now it needs a synchronizer. Moreover, this synchronizer should now a particular protocol to work with the system, that differs from simple round-begin/round-end notifications. One can argue that this actually brings the model to the asynchronous one because we define a way synchronizer works.

In this case, we have to take into consideration the communication complexity of centralized synchronization, here and further we should add $O(n)$ term for any place where we use phase transition. Theoretically, this can be improved by removing centralization and adding verification, that could be done in $\Omega(\sqrt{n} + D)$ time [51].

2.5.3 Blocking Flow Algorithm

Originally, blocking flow method was introduced in [52]. [17] suggests $O(n \log(n))$ time complexity algorithm for blocking flow calculation. A lower bound for parallel implementation was proved by [53]. Using PRAM without bit operations, the problem can be solved in $O(n^{1/8})$. [23] converges the shortest path in a finite number of steps.

This time was improved in [18] to $O(m)$, using DFS and exploiting the property of G_A that it is acyclic and contains unit capacities. Currently, this is the best upper bound for unit capacities. DFS can be easily run in distributed system using $O(diam)$ time and $O(m)$ message complexity.

2.5.4 Complexity

For sequential run the upper bound for time is [18]:

$$O(\min\{n,m^{1/2}U^{1/2},n^{2/3}U^{1/3}\}m \min\{\log n,U\} \log(nC))$$

Summarizing all above, for communication complexity, we have $O(\min\{m^{1/2},n^{2/3}\})$ steps for each iteration and $\log(nC)$ iterations for

distributed setting as well. At each iteration we have $O(D)$ messages for each DFS during a blocking-flow computation and $O(m)$ messages for BFS for each excess node. In the worst case, there can be $O(n)$ excesses, however for latest iterations another bound of

$$O(\min\{m^{1/2},n^{2/3}\})$$

on a number of excesses exist [18]. In total, for unit capacities, we get the communication complexity of

$$O(\min\{m^{1/2},n^{2/3}\}mDn\log(nC))$$

.

For the timing complexity the difference with the sequential estimation is in $O(D)$ for DFS run and $O(D)$ for BFS run. So, we have

$$O(\min\{m^{1/2},n^{2/3}\}D^2\log(nC))$$

If we consider syncronization cost, that does not influence time complexity, but would lead to additional n term to communication complexity:

$$O(\min\{m^{1/2},n^{2/3}\}mDn^2\log(nC))$$

3 Experiments

All experiments were conducted on Ubuntu 14.03 operating system, on a workstation with processor Intel(R) Xeon(R) CPU E5-2643 v2 @ 3.50GHz and 32G RAM. All software is written in C++.

3.1 Assignment problem and SIA

In this section, we compare the performance of SIA algorithm with LDA, DF-CSA, and BlossomV. We use our implementation of SIA and LDA, described in the Section 2. An efficient implementation of BlossomV is provided by the authors. DF-CSA implementation is part of LEMON library.

3.1.1 Complete Random Bipartite graphs

Complete Random Bipartite graph is expected to highlight the benefits of SIA. Tight threshold and high density of a graph lead to a high fraction of pruned edges and, as a result, high performance.

Data description The size of graphs for this experiment varies between 128 and 16 384 nodes. Each graph contains $\frac{n^2}{2}$ edges. SIA uses only a subset of that edges, according to the presented pruning technique.

Three types of distributions of edge weights are used. C++ Boost library is used as a random number generator.

1) Uniform distribution with weights in range 1 to 100

2) Gaussian distribution, PDF $f(x) = \frac{1}{\sigma\sqrt{1\pi}} \exp -\frac{(x-\mu)^2}{2\sigma^2}$, where $\mu = 30$ and $\sigma = 10$

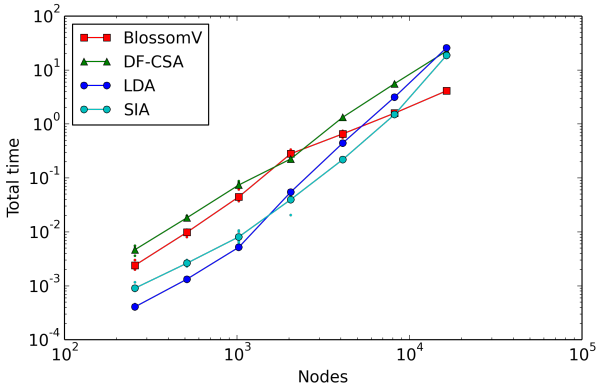
3) Exponential distribution, PDF $f(x) = \lambda e^{-\lambda x}$, where $\lambda = 0.01$

Results Figure 3.1 presents Total Execution time of 4 algorithms mentioned above. For different types of weight distribution, SIA tends to have the best time among exact solutions and almost the same time comparing to Approximate LDA.

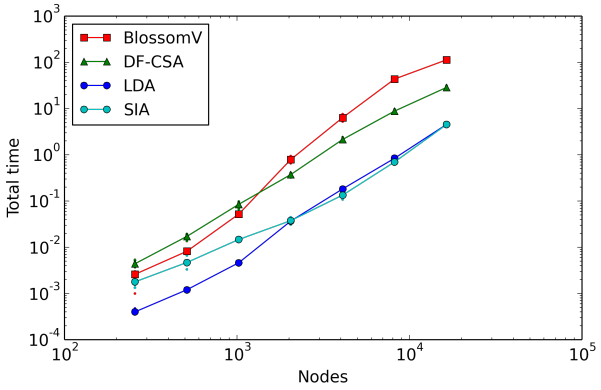
LDA shows better performance on small problems and it also has an advantage of much simpler implementation, but the analysis of the approximation error, presented on Figure 3.2, shows that LDA has a

significantly large error if a distribution is not Uniform. This shows that SIA can be useful even if there is no exactness requirement in a problem.

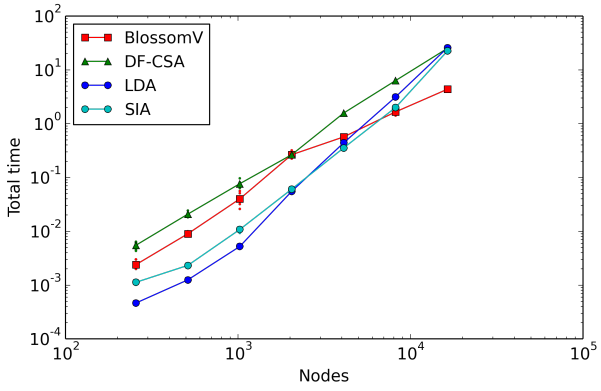
On large scales for Uniform and Exponential distributions, BlossomV surprisingly shows better scalability. However, the algorithm behaves in quite an unstable manner, as we can see that on Gaussian distribution it suddenly becomes extremely slow.



(a) Uniform weight distribution



(b) Gaussian weight distribution

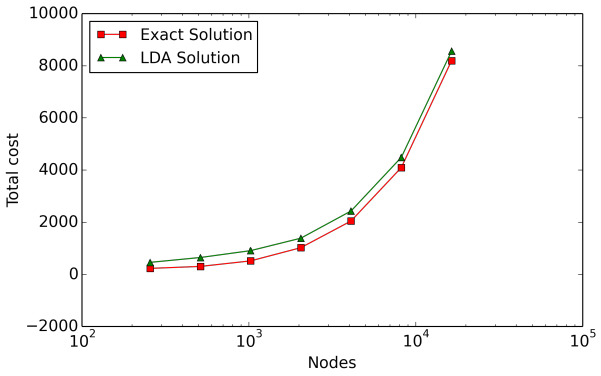


(c) Exponential weight distribution

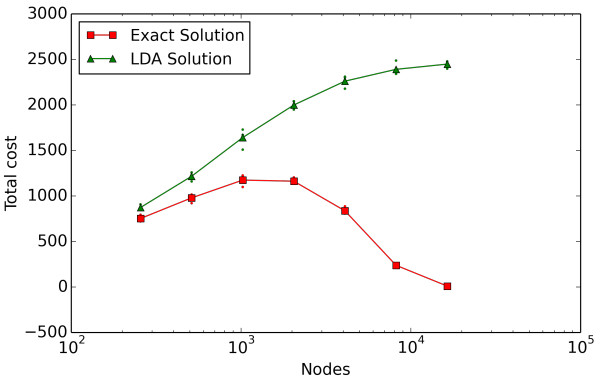
Figure 3.1 — Scalability of algorithms on Complete Bipartite graphs

The tendency of decrease of total cost for Gaussian and Exponential distributions are another remarkable result. If a problem is big enough, an exact algorithm can find a connection (edge) with minimal weight and for mentioned problems the minimum value of weight is equal to zero. So, total sum has a limit of 0 with bigger problem size as well. This also motivates to use exact solution for minimization problems, as well as half-approximation guarantee violation, presented in Section 1.

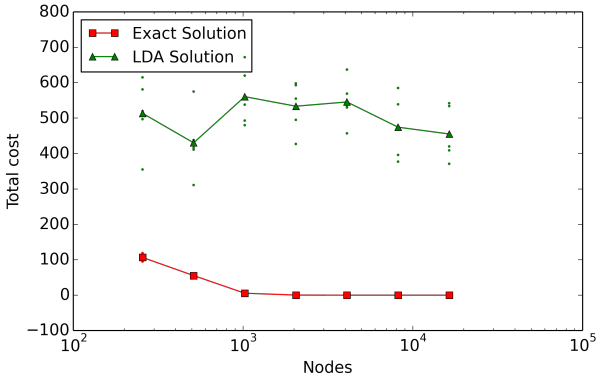
Spatial Data A geometric graph, where a weight of each edge is equal to the distance between nodes in space, has a distribution of edge



(a) Uniform weight distribution



(b) Gaussian weight distribution



(c) Exponential weight distribution

Figure 3.2 — Approximation error of LDA

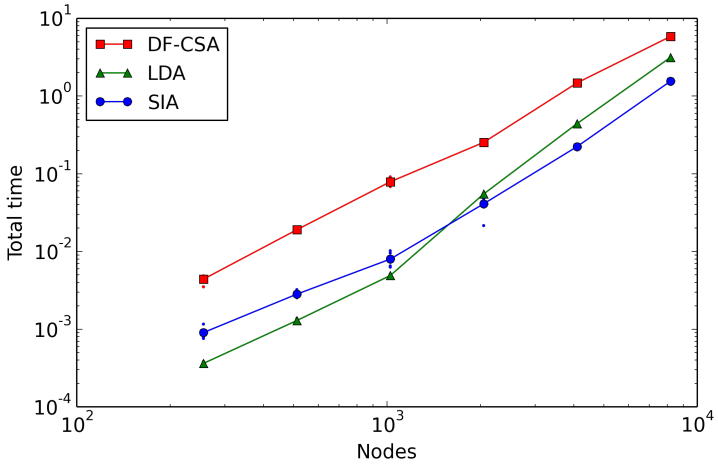


Figure 3.3 — Performance of SIA, LDA and DF-CSA on Bipartite Uniform Spatial data

weights, illustrated on Figure 3.4. The results are illustrated on Figure 3.3. The relative performance of SIA is comparable with results on Complete Random Bipartite graphs, that is in correlation with the fact of close similarity of distributions. But additionally to the time superiority, SIA takes advantage out of space benefits for large problems, since there is no need of holding complete graph in the memory.

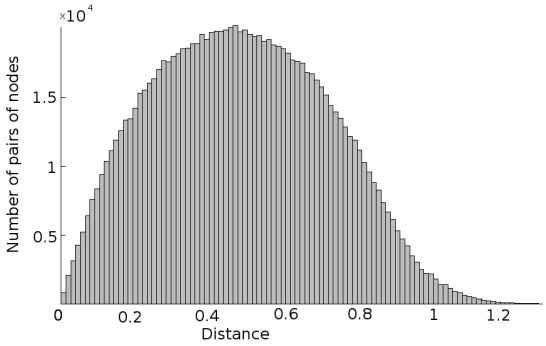
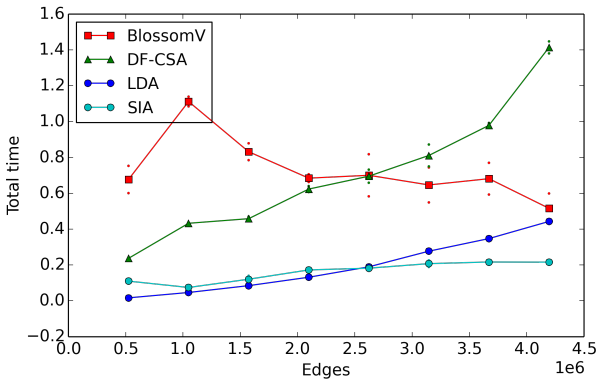


Figure 3.4 — Distribution of weights of edges in geometric graph based on random point distribution

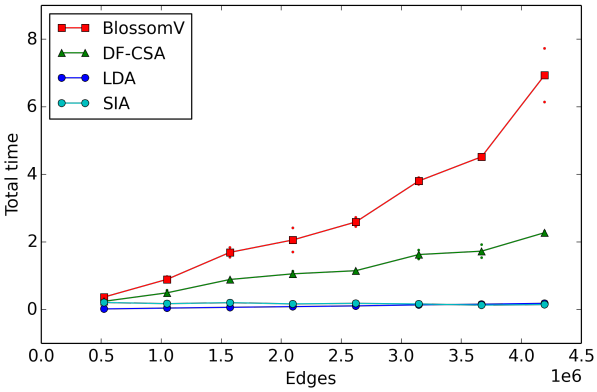
3.1.2 Sparse graphs

The influence of graph sparsity is illustrated on Figure 3.5. For a complete bipartite graph, some random edges are removed. At each experiment, we remove a certain number of random edges equally for each node. The total resulting number of edges is a variable on the plot. As a result, we can observe a dependence on the sparsity of the graph. Three types of weight distribution are used, the same as in the previous section.

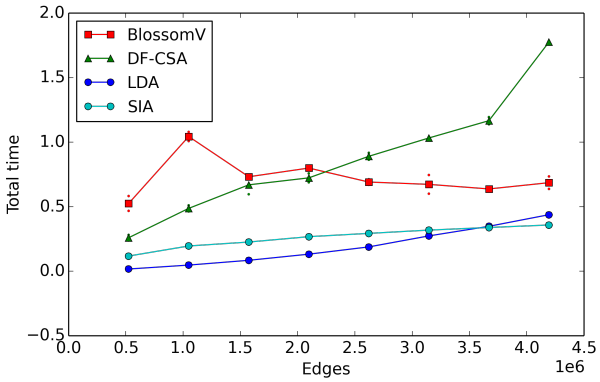
For all three distributions, SIA shows excellent results with a total time almost the same as an approximate algorithm. No significant dependence on graph sparsity is observed. DF-CSA is distinctive by its stability and equal performance for any distribution, however, the total time linearly increases. BlossomV, as in the previous section, show bad performance on Gaussian weight distribution. Probably, the quality of the algorithm depends on the possibility to collapse blossoms, so we can see a high peak on the first and third plots, and there are no satisfactory conditions to do that on the second.



(a) Uniform weight distribution



(b) Gaussian weight distribution

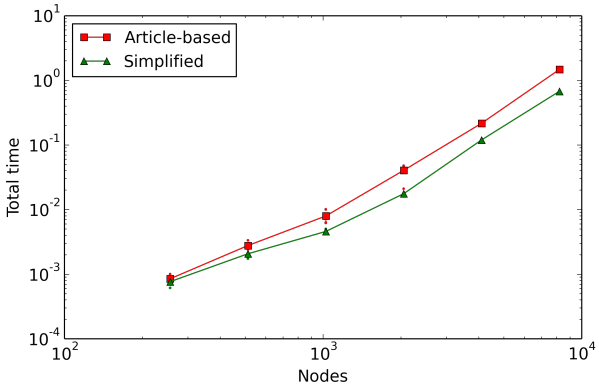


(c) Exponential weight distribution

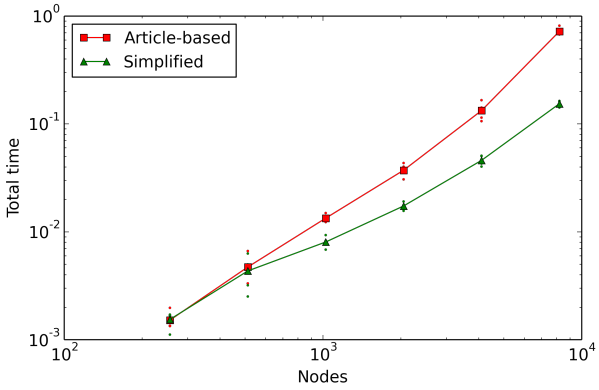
Figure 3.5 — Dependence of bipartite graph sparsity on the performance of matching algorithms

3.1.3 Heap value variation

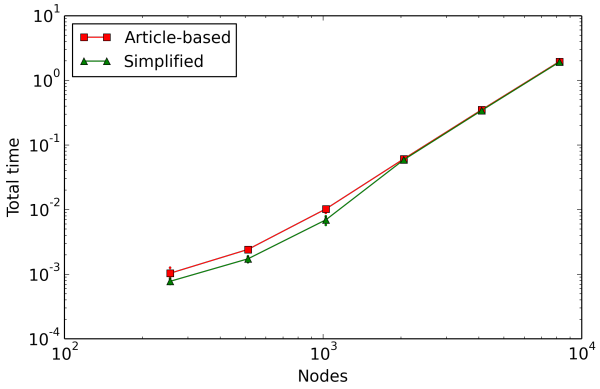
In this section, we compare two options for a pruning threshold. Article-based is a threshold, described in the Section 1.3. Simplified is a novel approach, described in Section 2.2. Figure 3.6 shows that on complete bipartite graph Simplified version runs faster on all types of weight distributions, especially for Gaussian.



(a) Uniform weight distribution



(b) Gaussian weight distribution



(c) Exponential weight distribution

Figure 3.6 — Performance of novel heap threshold

3.2 DF-CSA analysis

First, we test sorting and pruning on complete bipartite graphs. This time, we consider all three types of distribution together. Figure 3.7b shows that there is no significant improvement on performance, especially on larger scales.

Figure 3.8 shows the fraction of pruned edges for S-DF-CSA. The data used is randomly distributed points in space. Every two points allow

bidirectional transfer of a flow, i.e. in the graph representation for each pair of nodes there are two edges in both directions between them. Every line on the figure represent the total amount of nodes in the graph (3.8a) or number of source nodes in the graph of size 8192 nodes (3.8b). A number of target nodes are always greater than a number of source nodes in this experiment. We calculate the saturation of the graph as an average fraction of a number of edges that have been traversed through the algorithm divided by the number of total edges for each node.

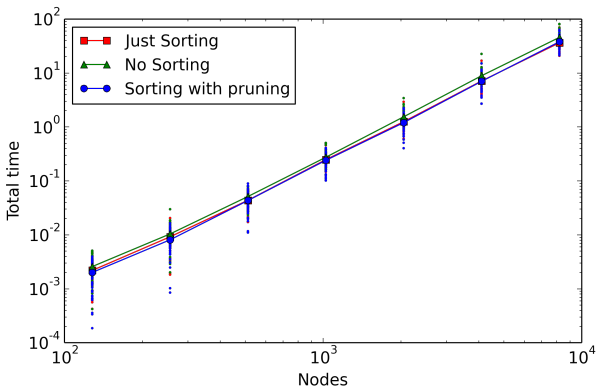
$$Saturation = \frac{1}{n} \sum_{v \in V} \frac{|\{(v,w) \in E_{sub}\}|}{|\{(v,u) \in E\}|}$$

The absolute amount of source nodes varies from half of the size of the graph down to 1/128 of the size, decreasing in geometric order every power of 2.

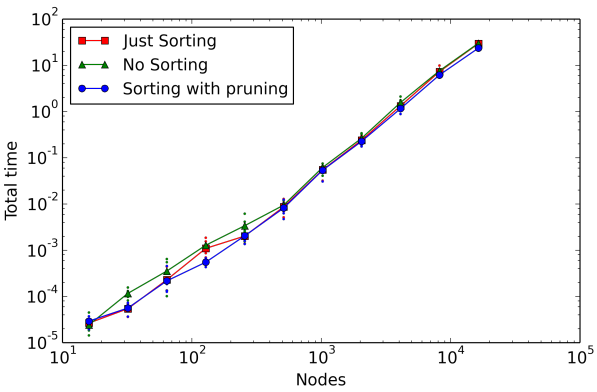
Results show that there is only slight dependence for uniform distribution between edge saturation of the graph and the ratio. In general, our method allows pruning 10-30% of edges for the uniform distribution. Larger scales lead to the smaller amount of pruned edges, partially because an absolute number of source and target nodes is larger, so they evenly cover a larger fraction of a space. Smaller graphs allow pruning more data, for some, the ratio even drops below 30%. The bipartite case is a special case of unit ratio, and for most graph sizes saturation does not differ much even from a single-source case.

3.2.1 Heuristics influence

As was mentioned above, DF-CSA uses several heuristics that improves the solution. In this work we develop optimized algorithms that are based on CSA, but in order to simplify our work, we omit embedding of that heuristics in our solutions. Several experiments were conducted in order to check the significance of those heuristics and a possibility to remove them while designing new algorithm. Results are illustrated on fig. 3.9. As we can see that the impact is not significant, we leave embedding of heuristics in our approach as a future work.

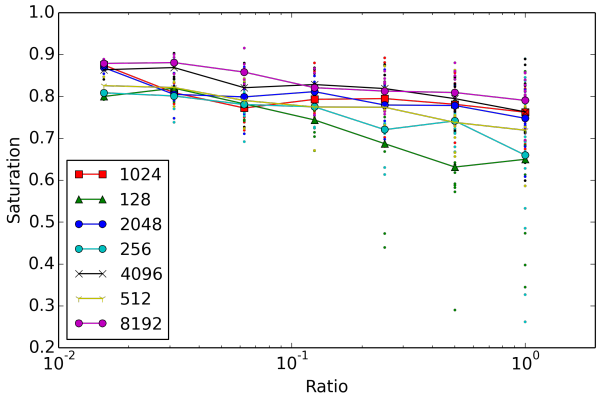


(a) Randomly distributed points in space, all fractions of source/target nodes

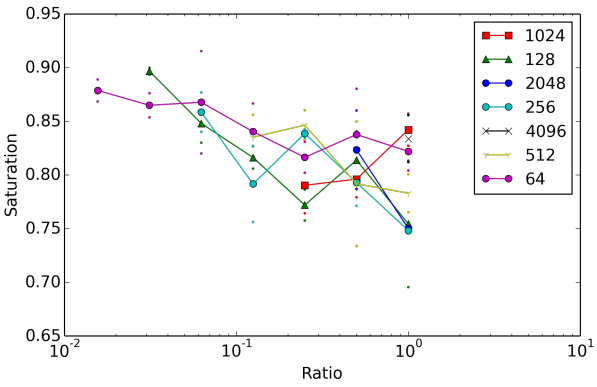


(b) Complete bipartite graph (all types of weight distributions)

Figure 3.7 — Influence of sorting and pruning on DF-CSA performance



(a) Each line represent the size of the graph



(b) Each line represents number of source nodes, the size fixed on 8192 nodes

Figure 3.8 — Dependence of the fraction of non-pruned edges on the ratio *sources/targets* in S-DF-CSA for randomly distributed spatial points

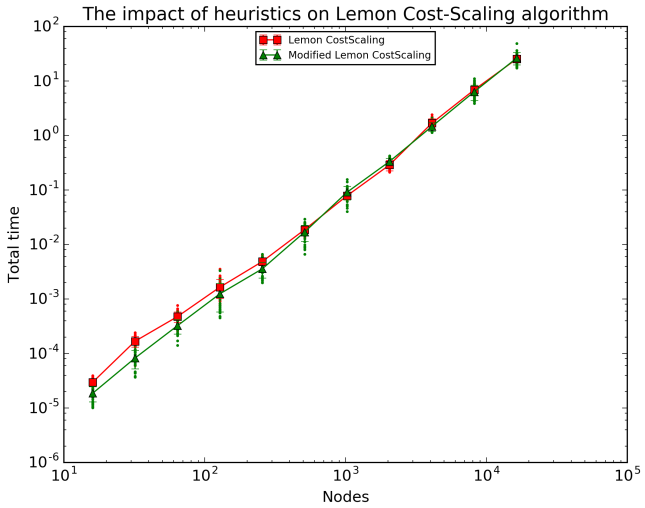


Figure 3.9 — The impact of heuristics on DF-CSA on random full bipartite graphs

For a graph of small diameter, like the Internet topology graph from SNAP library [54] by Skitter ($1.7 \cdot 10^6$ nodes, $11 \cdot 10^6$ edges, Longest path: 25), the heuristic has an opposite effect: if all paths between sources and destinations are very short, then the usage of heuristics slows down the algorithm. For the mentioned graph for randomly distributed sources and targets, it takes about 5 seconds to finish Lemon-based implementation and less than one second to run DF-CSA without any heuristic.

3.3 S-CSA analysis

In this section, we investigate the performance of 3 variations of S-CSA on spatial uniformly distributed random points. Each point has a possibility to send flow to any other point in the space. 3 variations include (1) decreasing ϵ from the size of the universe (maximum available distance between any pair of points) to 1; (2) holding $\epsilon = 1$ throughout the algorithm; (3) the same as in (1), but with disabled pruning. We call the variations "Large epsilon "Unit epsilon"and "No pruning accordingly. Note, that the first variation is S-CSA as described in Algorithm 7, the second variant is a simple generalization of SIA on general graphs, and the third is our implementation of the classical Cost-Scaling algorithm with bucket-based Dijkstra and graph representation described in the previous Chapter.

3.3.1 Graph traversal

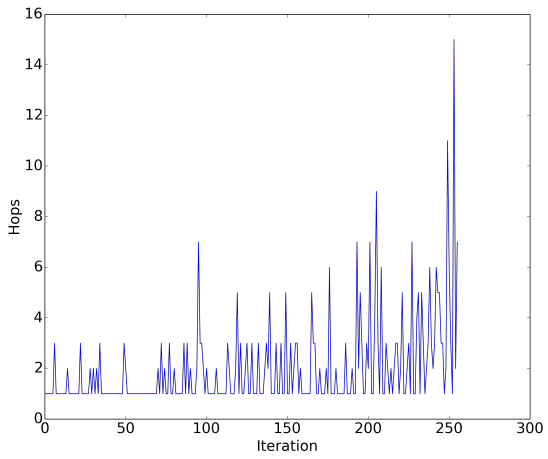
In this section, we investigate the behavior of Dijkstra algorithm in different variations of the algorithm. The length of the shortest path in Unit Epsilon variation os S-CSA is illustrated on Figure 3.10. First of all, we consider a bipartite case (subfigures (a),(b) and (e)). SIA has the same behavior since Unit epsilon case applied on a bipartite graph is almost the same algorithm as SIA. The difference is only in a subset of edges where a maximum potential is calculated from, and missing H_{upd} in S-CSA. We can see that majority of iterations stops after 1-2 hops and when all possibilities to assign one of the first nearest neighbors are utilized, then the algorithm starts to reassign nodes. On Figure 3.10a there is an increase in hops closer to the end of the algorithm. The increase of the shortest path cost on Figure 3.10b also confirms the late reassignment.

Spatial data with all possible flow directions does not have such property. Short and long shortest paths are alternating, together with a total path cost (Figures 3.10d, 3.10c). The magnitude is also much bigger in average than in the bipartite case. However, in the majority of iterations, the length is still about one, so there is a potential for a pruning.

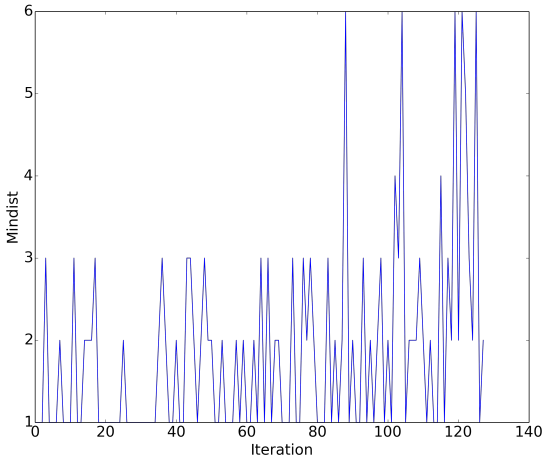
3.3.2 Spatial uniformly distributed random data

The experiment conducted for a different amount of sources, targets and graph sizes. A graph for this problem set is a clique. For each graph size number N of sources vary from 128 to the half of a graph size each power of 2. We refer to the number of sources as S. For each S, number of targets T vary from T to $N/2$. Supply value for each target is 1 and for each source, it is equal to S/T, i.e. a flow should be equally distributed between sources. While describing results, we assume that if a parameter is not set, then all possible values are used. For example, if a plot illustrates Total time depending on S for $N = 128$, then for each S all possible T are used.

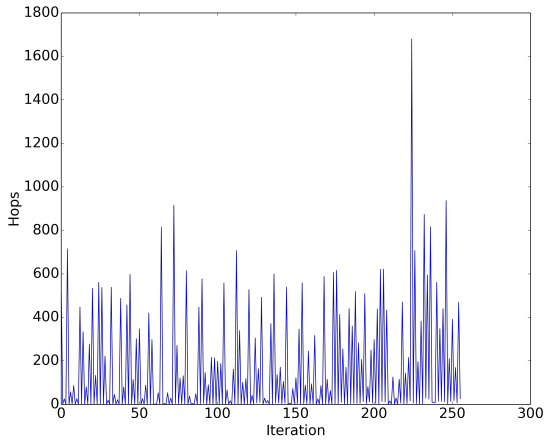
The results are shown of Figures 3.12 and 3.11. Figure 3.11 shows the dependence of algorithm time on number of target nodes. Algorithm time is a total running time without time for finding next nearest neighbor. Last



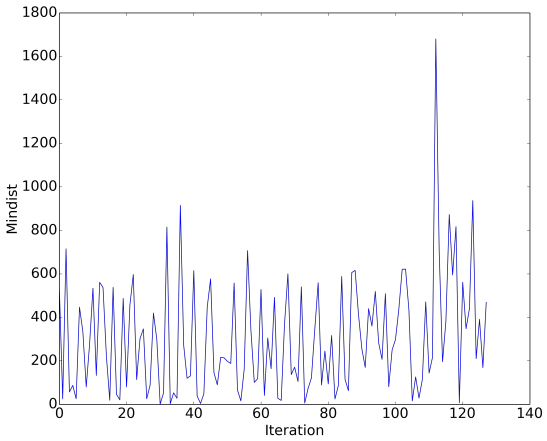
(a) A length (hops) of a shortest distance at each iteration (bipartite)



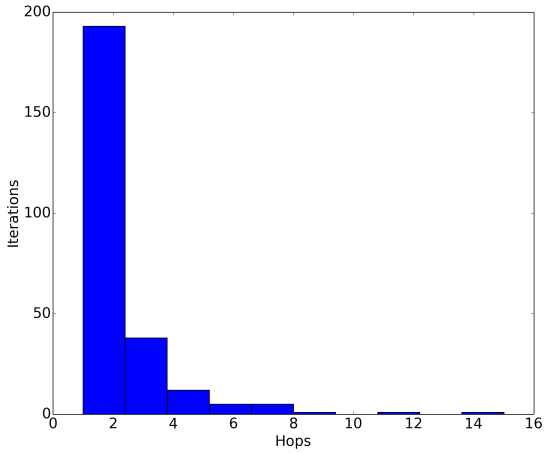
(b) A minimum distance to a target at each iteration (bipartite)



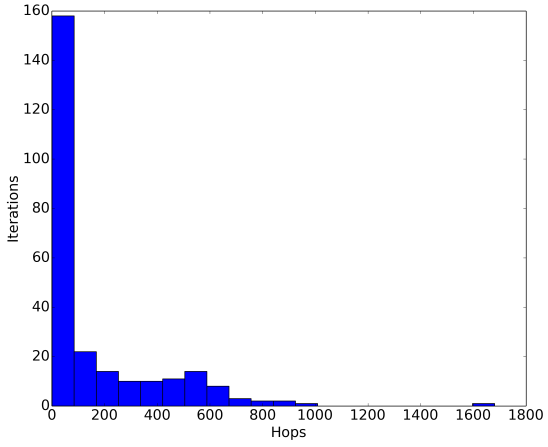
(c) A length (hops) of a shortest distance at each iteration (spatial)



(d) A minimum distance to a target at each iteration (spatial)



(e) A histogram of length of all shortest paths throughout the execution (bipartite)



(f) A histogram of length of all shortest paths throughout the execution (spatial)

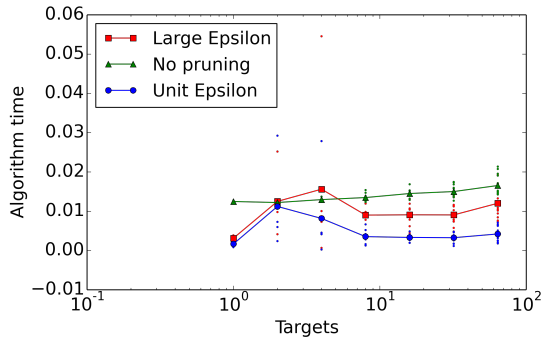
Figure 3.10 — A length and minimum distances during Dijkstra execution on complete bipartite random graph of size 512 with uniform weights, and on randomly distributed spatial points of size 1024 with 32 sources and 128 targets

two plots show total running time as well. The time of S-CSA with Unit Epsilon overcomes S-CSA with Large Epsilon in all experiments of this set. Also, Large epsilon has a very significant dispersion of the time. For unlucky distributions, the time can rapidly increase several times, so the solution turned out to be quite unstable for different source and target distributions. The S-CSA with no pruning behaves differently comparing to other variations depending on the size of the problem. For smaller sizes, S-CSA with Unit Epsilon shows the best time, but for larger problems classical cost-scaling approach is better. This correlates with the number of pruned edges, that is larger for small graphs. However, last two plots with total running time show that for randomly distributed points in sequential setting the time for adding edges does not radically change the ratio of performance between different algorithm variations. The ratio is nearly the same according to the Algorithm time, despite the fact that the time for adding new edges increases the total running time almost twice.

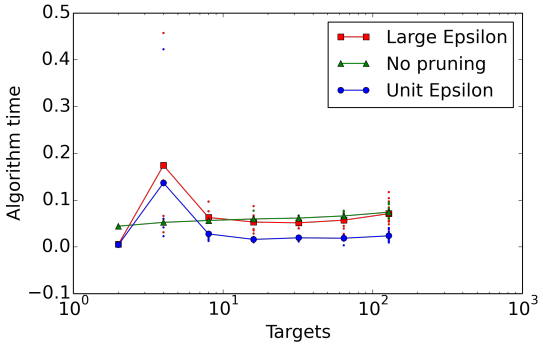
Figure 3.12 is similar to the previous one. It illustrates the dependence of algorithm time on a number of source nodes. The general tendency is the same. Also, we can mention that experiments did not reveal any noticeable dependence of the performance on the number of sources or targets. Last two subplots on the figure illustrate the dependence on the ratio between sources and targets. A slight dependence is visible, but it is not significant and mainly describes the increase in the running time if a sum of source and target nodes is bigger.

3.3.3 Clustered spatial points

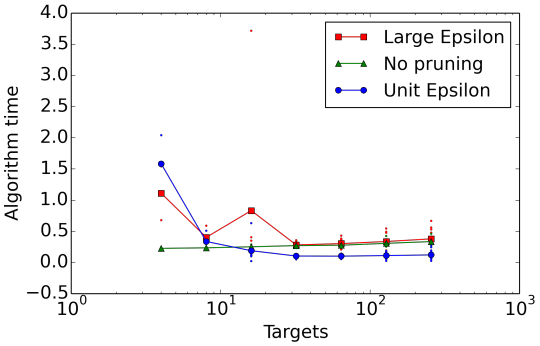
In order to test the difference between $\epsilon(T)$ functions on more complex topology, we generated clustered spatial points with a different number of sources S, targets T, total points N and a number of clusters C. Examples of data are illustrated on the Figure 3.13. Results of the experiment are combined in two Figures 3.14 and 3.15. Figure 3.14 shows the total time of the algorithm depending on different variables. Figure 3.15 shows the saturation of the graph, i.e. number of edges added divided by the total available edges.



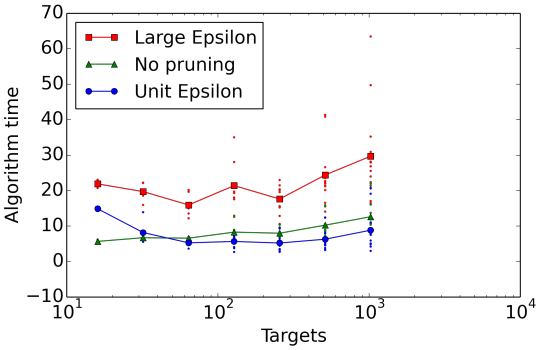
(a) $N = 128$



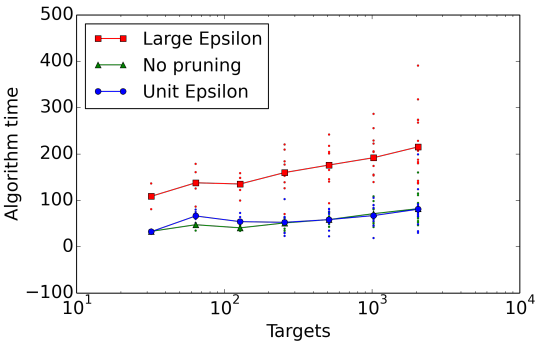
(b) $N = 256$



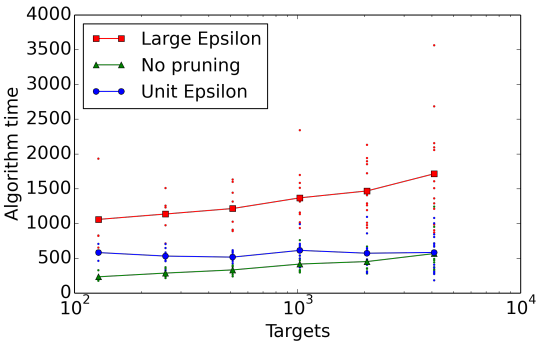
(c) $N = 512$



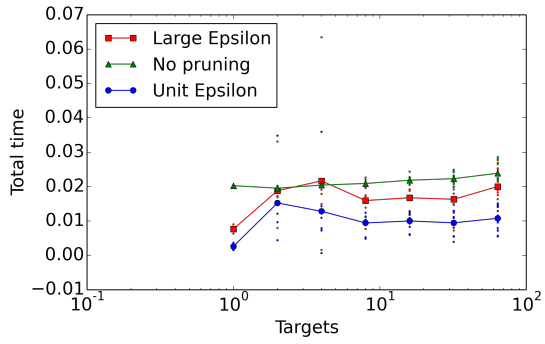
(d) $N = 2048$



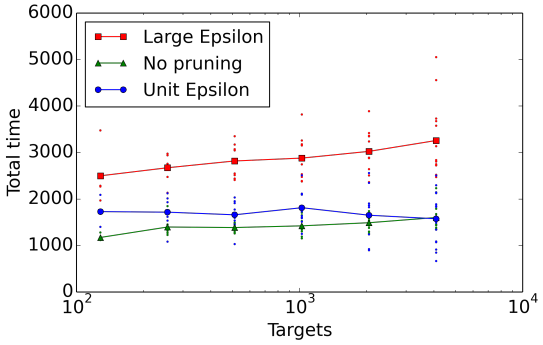
(e) $N = 4096$



(f) $N = 8192$

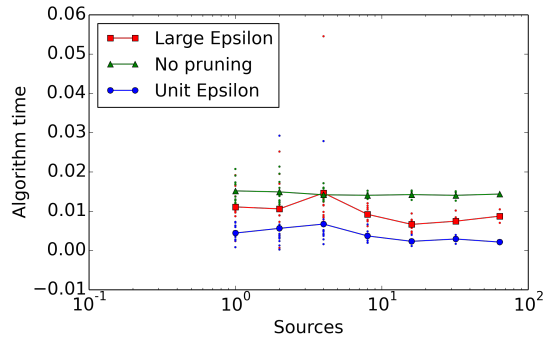


(g) Time with edge addition $N = 128$

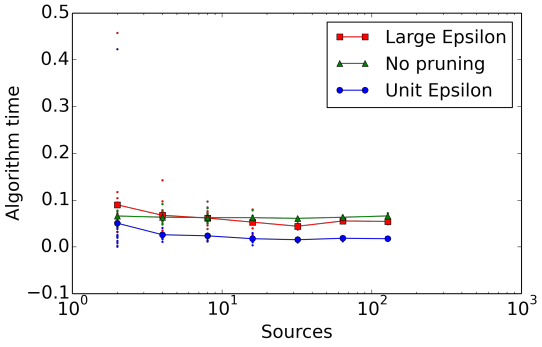


(h) Time with edge addition for $N = 8192$

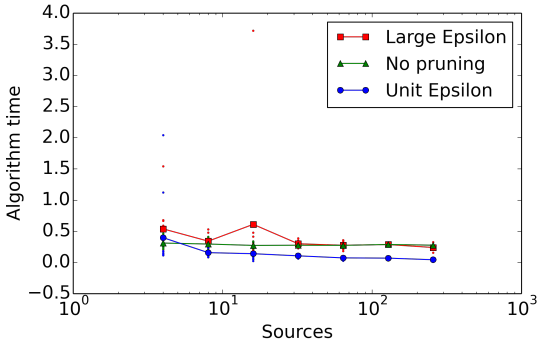
Figure 3.11 — Dependence of total running time without edge addition time on T for N



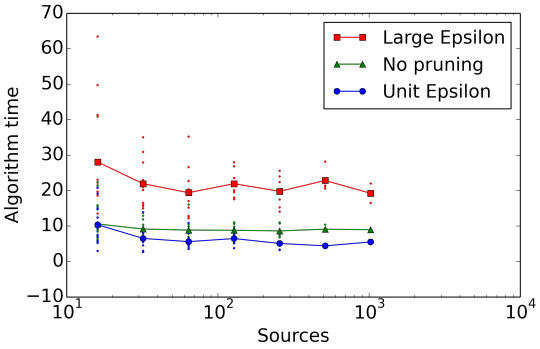
(a) $N = 128$



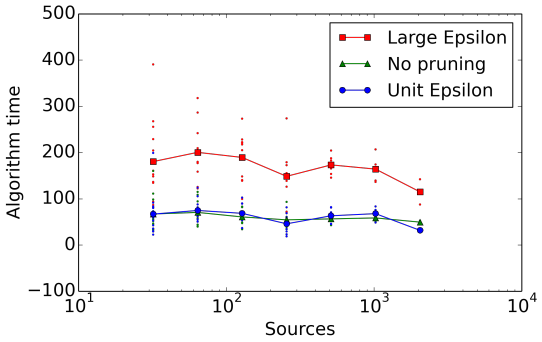
(b) $N = 256$



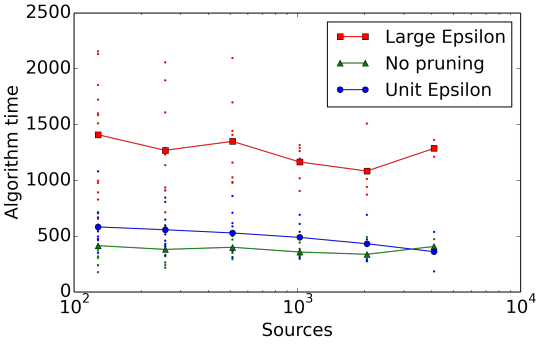
(c) $N = 512$



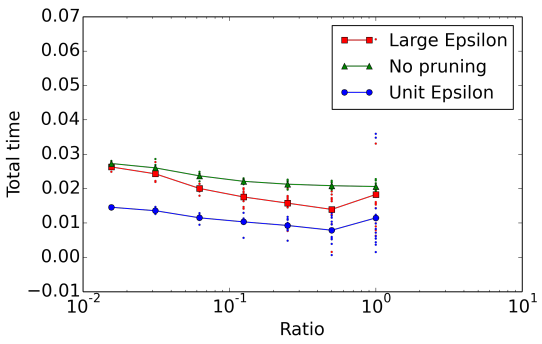
(d) $N = 2048$



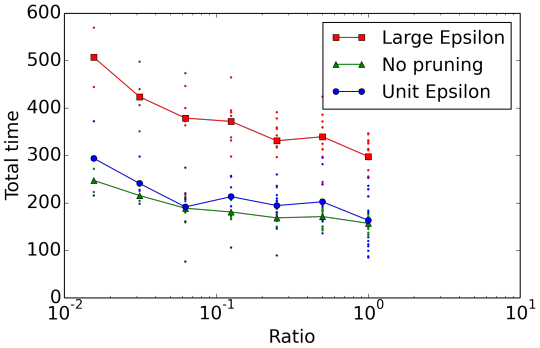
(e) $N = 4096$



(f) $N = 8192$



(g) Parameter: ratio $\frac{S}{T}$ for $N = 128$



(h) Parameter: ratio $\frac{S}{T}$ for $N = 4096$

Figure 3.12 — Dependence of total running time without edge addition time on S for N

We vary every mentioned parameter in order to find a range of parameters where each variation of the algorithm gives the best relative performance.

As we can see on the plots, in general, the fraction of pruned edges is about 0.5 for small problems sizes or small source and target nodes. For a more dense fraction of sources and targets, all edges become covered and the benefit from pruning vanishes. Total time reflects this, S-CSA with pruning outperforms pure ϵ-scaling on smaller problems. Another note is that unfortunately, Large Epsilon modification loses comparing to Unit epsilon almost on each sample. Unit epsilon naturally should have a larger fraction of pruned edges since potential values are tighter, and the benefits from cost scaling can not beat that. Also, it is important to mention the huge variation of total time of S-CSA with pruning. It greatly depends on the topology of the graph.

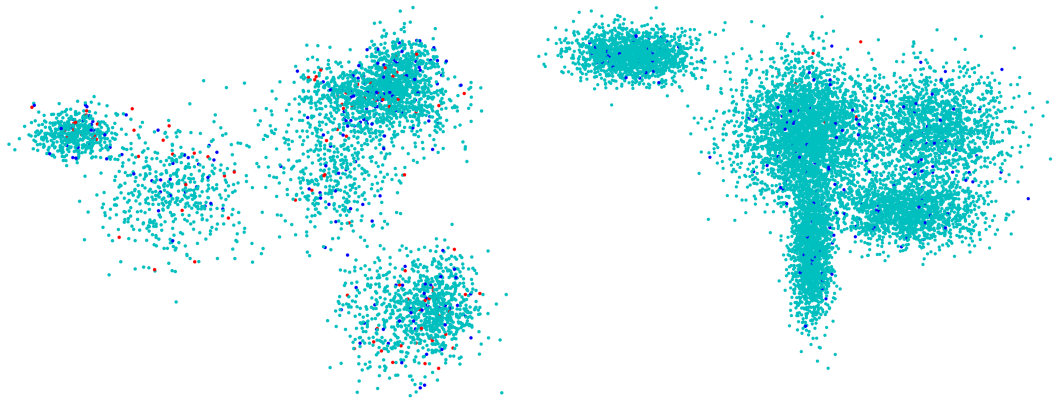
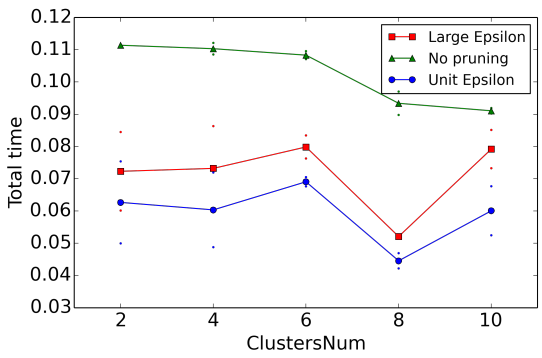


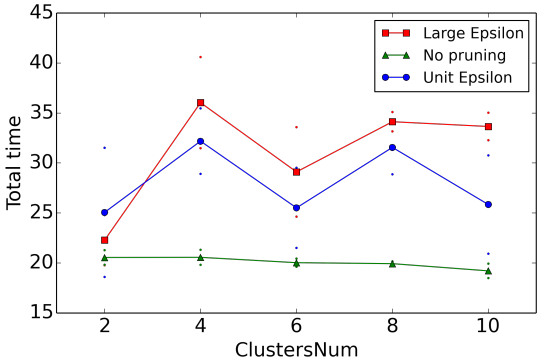
Figure 3.13 — Examples of clustered points. Red points - sources, green - targets, blue - neutral

3.4 Power Flow optimization

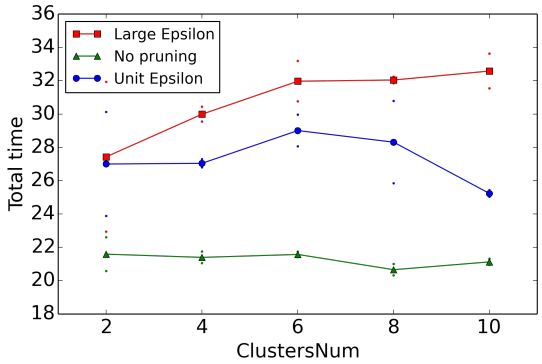
In order to test our algorithms on real-world data, we used the graph of power plants and cities in China. The data was obtained from The China Climate and Energy Map, a project of the Natural Resources Defense Council's China Program `http://www.chinaenergymap.org/`. Instead of solving Optimal Power Flow problem, we solve a problem of finding optimal power grid over the country, such that all capacities of power plants and



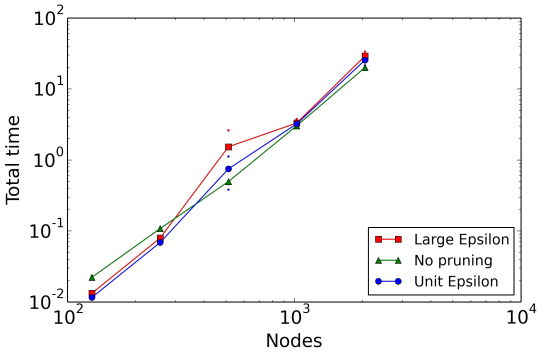
(a) Time(C), $N = 256$, $S = T = 64$



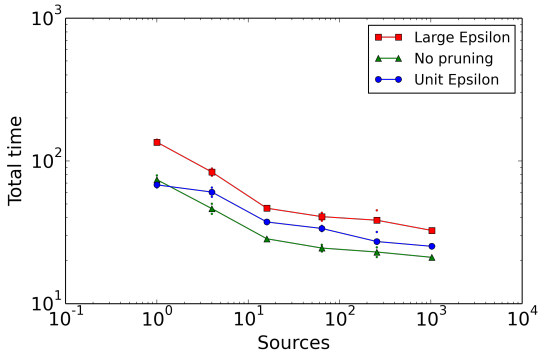
(b) Time(C), $N = 2048$, $S = T = 64$



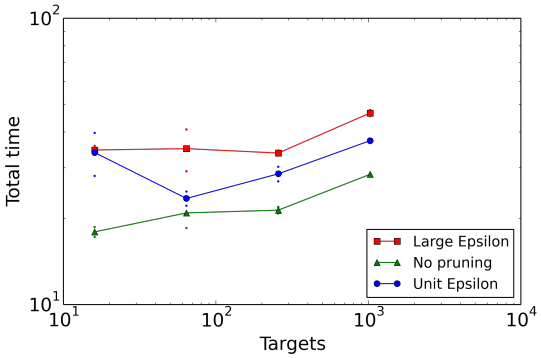
(c) Time(C), $N = 2048$, $S = T = 1024$



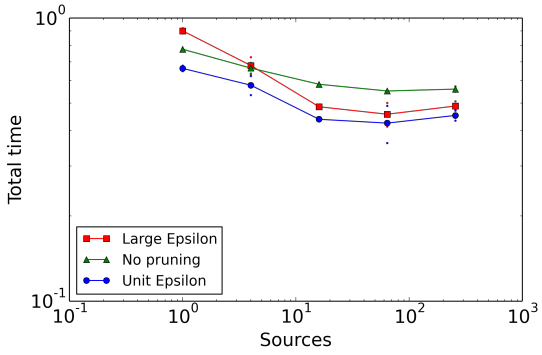
(d) Time(N), $C = 6$, $S = T = 64$



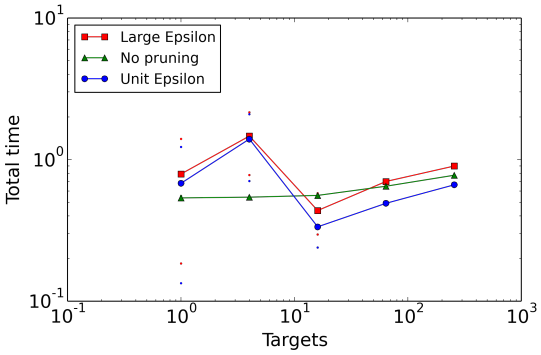
(e) Time(S), $N = 2048$, $C = 10$, $T = 1024$



(f) Time(T), $N = 2048$, $C = 10$, $S = 16$

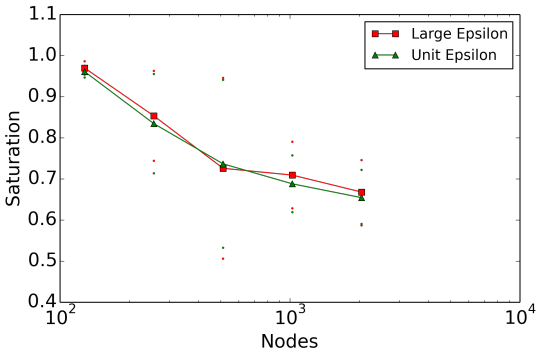


(g) Time(S), $N = 512$, $C = 2$, $T = 256$

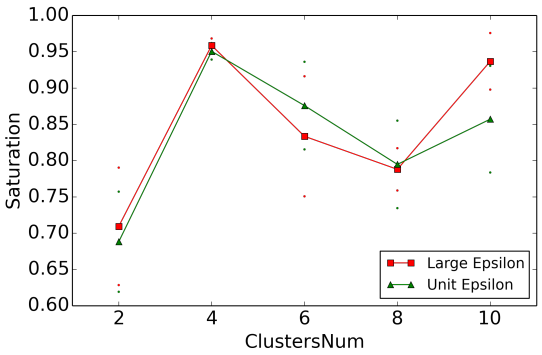


(h) Time(T), $N = 512$, $C = 2$, $S = 1$

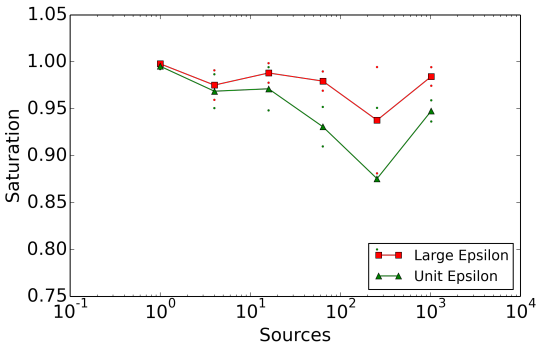
Figure 3.14 — Dependence of total running time on S, T, C, N



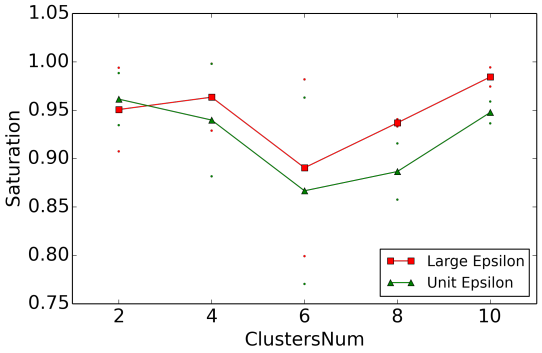
(a) Saturation(N), $C = 2$, $S = T = 64$



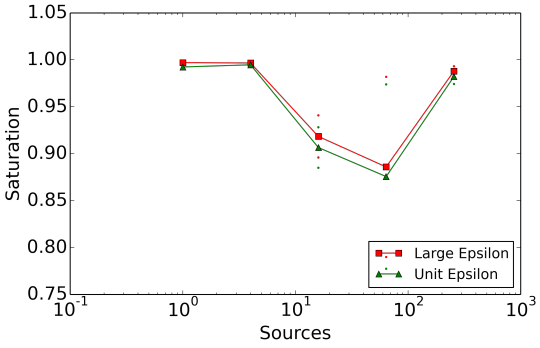
(b) Saturation(C), $N = 1024$, $S = T = 64$



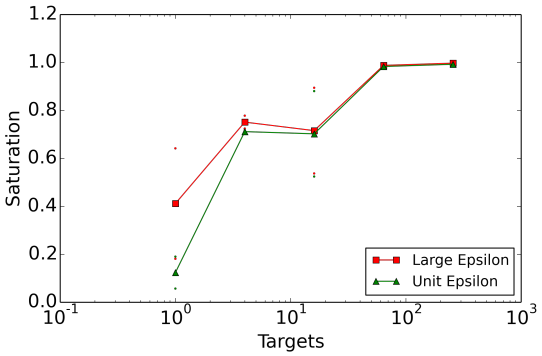
(c) Saturation(S), $N = 2048$, $C = 10$, $T = 1024$



(d) Saturation(C), $N = 2048$, $S = T = 1024$



(e) Saturation(S), $N = 512$, $C = 2$, $T = 256$



(f) Saturation(T), $N = 512$, $C = 2$, $S = 1$

Figure 3.15 — Dependence of Saturation on S, T, C, N

demands of large cities were satisfied. Given a power consumption of a region and its population, we assume that there is a linear dependency between these two parameters, see Figure 3.17. The Energy Map provides data about the population of major cities and the capacities of Coal, Nuclear, Gas and Hydro power plants. We build the graph in such a way that each plant is a spatial point with zero supply, each city is a point with negative supply, and one auxiliary non-spatial node is connected to each plant with a zero-cost edge and upper flow bound equal to the capacity of that plant. The resulting graph is illustrated on Figure 3.18. In order to make the problem less straightforward, we add non-linearity in a sense that the cost of each unit of flow between two points is quadratically dependent on the distance between them. Note here, that the framework of our algorithms allow any monotonically increasing function to be applied to the distance, since the only place in any pruning algorithm where a distance is used is sorting. While sorting is feasible, an algorithm performs correctly. The data contain 1992 points. DF-CSA process mentioned graph is 16.23 seconds and S-DF-CSA is 5.61 seconds (Figure 3.16).

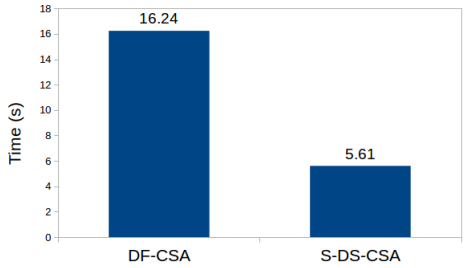


Figure 3.16 — The performance of algorithms on Chinese Energy Map

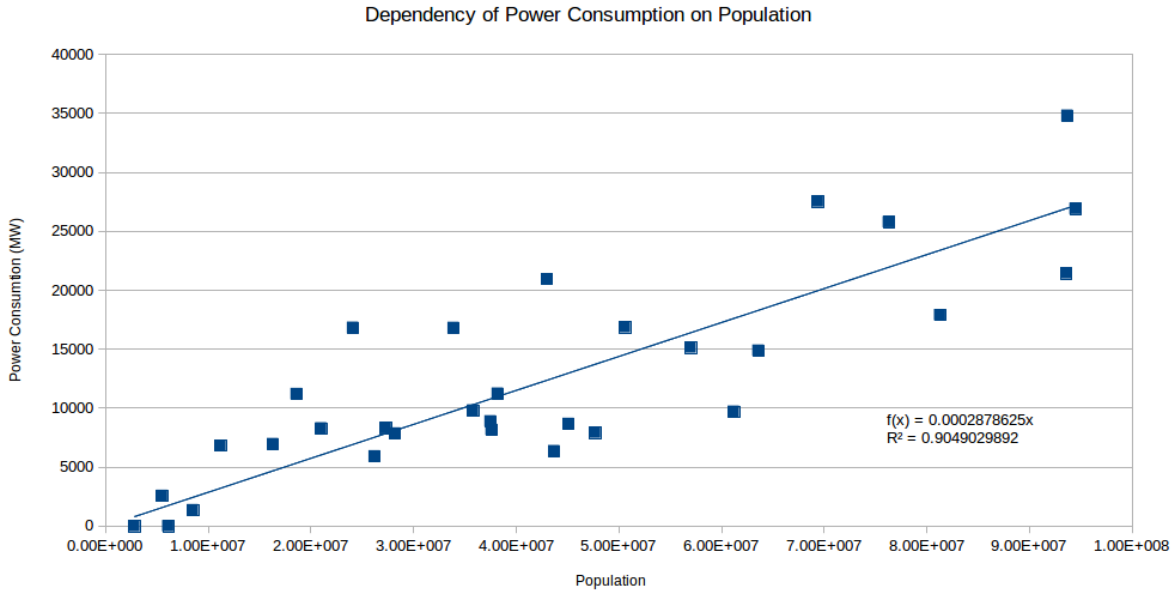


Figure 3.17 — The correlation between power consumption and population

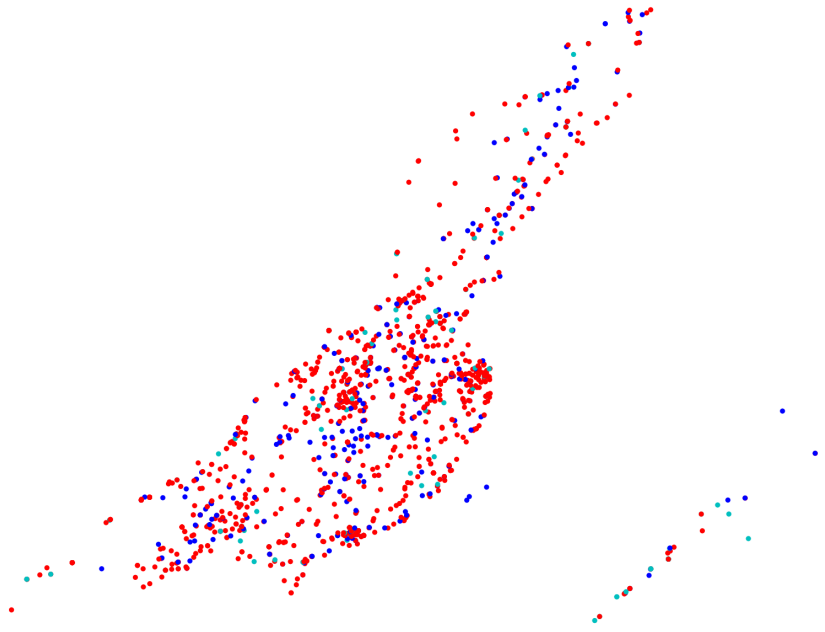


Figure 3.18 — Chinese energy map. Red - power plants, green, blue - cities

Conclusions

In this thesis we proposed several algorithms for Circulation Problem and Bipartite Matching problem, and investigated their performance on different types of data, in particular spatial data. They include a better threshold for SIA, that provide a slight improvement on complete bipartite graphs, a modification of DF-CSA and a novel S-CSA, that combined cost scaling technique and a pruning method of SIA. Unfortunately, ϵ-scaling does not fit well with pruning. The possible explanation behind this may be the intuition that both methods solve a problem by incremental increasing of an accuracy of current solution, but do it in complete opposite ways. The pruning method is based on increasing the number of used edges, that are sorted, i.e. increasing the accuracy by incremental exploring farther points. ϵ-scaling does the opposite. It increases the accuracy by first observing farthest points, and then after it finds an approximate coarse-grained solution, it starts to investigate fine-grained opportunities to improve the solution. On the other hand, ϵ is decreasing over the time and for some topologies of a graph the pruning gives a good gain in time.

Comparison of S-CSA with Large epsilon and Unit Epsilon clearly showed that Unit Epsilon performs better. Naturally, Unit Epsilon leads to a higher fraction of pruned edges, since the potential values are always tight. But Large Epsilon may increase Dijstkra execution on the earlier stages of the algorithm, so the benefit of Unit Epsilon was not quite clear. It turned out that even small growth of pruned edges can give more benefit to the execution time than reducing shortest path length by ϵ-approximation.

Pruning technique showed better results than classical Cost Scaling approach on many samples, mainly on small graphs or graphs with small amount of sources and targets. However, it lacks the stability and greatly depends on the topology. Bipartite case is still the best option to use pruning, that was also confirmed by the comparison with other state-of-the-art algorithms.

References

1. *Kolmogorov, Vladimir.* Blossom V: a new implementation of a minimum cost perfect matching algorithm / Vladimir Kolmogorov // *Mathematical Programming Computation.* — 2009. — Vol. 1, no. 1. — Pp. 43–67.

2. *Morteza Fayyazi David Kaeli, Waleed Meleis.* Parallel Maximum Weight Bipartite Matching Algorithms for Scheduling in Input-Queued Switches / Waleed Meleis Morteza Fayyazi, David Kaeli // *IEEE.* — 2004.

3. Quantifying the benefits of vehicle pooling with shareability networks / Paolo Santi, Giovanni Resta, Michael Szell et al. // *Proceedings of the National Academy of Sciences.* — 2014. — Vol. 111, no. 37. — Pp. 13290–13294.

4. Optimal matching between spatial datasets under capacity constraints / Leong Hou U, Kyriakos Mouratidis, Man Lung Yiu, Nikos Mamoulis // *ACM Transactions on Database Systems (TODS).* — 2010. — Vol. 35, no. 2. — P. 9.

5. *Hopcroft, JE.* An n$\hat{5}$/2 algorithm for maximum matchings in bipartite graphs / JE Hopcroft // *SIAM Journal on computing.* — 1973.

6. *Bertsekas, Dimitri P.* Auction algorithms for network flow problems: A tutorial introduction / Dimitri P. Bertsekas // *Computational Optimization and Applications.* — 1992.

7. *Duan, Ran.* Scaling algorithms for approximate and exact maximum weight matching / Ran Duan, Seth Pettie, Hsin-Hao Su // *arXiv preprint arXiv:1112.0790.* — 2011.

8. Near-Optimal Distributed Maximum Flow / Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn et al.

9. Communication Complexity of Approximate Matching in Distributed Graphs / Zengfeng Huang, Božidar Radunović, Milan Vojnović, Qin Zhang // 32nd International Symposium on Theoretical Aspects of Computer Science. — 2015. — P. 460.

10. *Dijkstra, E. W.* A note on two problems in connexion with graphs / E. W. Dijkstra // *Numerische Mathematik.* — Vol. 1. —

P. 269–271.

11. *Panitanarak, Thap.* Performance analysis of single-source shortest path algorithms on distributed-memory systems / Thap Panitanarak, Kamesh Madduri // CSC14: The Sixth SIAM Workshop on Combinatorial Scientific Computing. — 2014. — P. 60.

12. *Meyer, Ulrich.* Δ-stepping: a parallelizable shortest path algorithm / Ulrich Meyer, Peter Sanders // *Journal of Algorithms.* — 2003. — Vol. 49, no. 1. — Pp. 114–152.

13. *Kuhn, Harold W.* The Hungarian method for the assignment problem / Harold W Kuhn // *Naval research logistics quarterly.* — 1955. — Vol. 2, no. 1-2. — Pp. 83–97.

14. *Ford, Lester R.* Maximal flow through a network / Lester R Ford, Delbert R Fulkerson // *Canadian journal of Mathematics.* — 1956. — Vol. 8, no. 3. — Pp. 399–404.

15. *Derigs, Ulrich.* A shortest augmenting path method for solving minimal perfect matching problems / Ulrich Derigs // *Networks.* — 1981. — Vol. 11, no. 4. — Pp. 379–390.

16. *Edmonds, Jack.* A glimpse of heaven / Jack Edmonds // *History of Mathematical Programming: A collection of Personal Reminiscences (JK Lenstra, AHG Rinnoy Kan and A. Schrijver eds.), North-Holland.* — 1991. — Pp. 32–54.

17. *Goldberg, Andrew V.* Finding minimum-cost circulations by successive approximation / Andrew V Goldberg, Robert E Tarjan // *Mathematics of Operations Research.* — 1990. — Vol. 15, no. 3. — Pp. 430–466.

18. *Andrew V. Goldberg Haim Kaplan, et al.* Minimum cost flows in graphs with unit capacities / et al. Andrew V. Goldberg, Haim Kaplan // 32nd International Symposium on Theoretical Aspects of Computer Science. — 2015. — P. 460.

19. *Cunningham, William H.* A network simplex method / William H Cunningham // *Mathematical Programming.* — 1976. — Vol. 11, no. 1. — Pp. 105–116.

20. *Kovacs, Peter.* Minimum-cost flow algorithms: an experimental evaluation / Peter Kovacs // *Optimization Methods and Software.* —

2015. — Vol. 30. — Pp. 94–127.

21. *Henzinger, Monika.* An Almost-Tight Distributed Algorithm for Computing Single-Source Shortest Paths / Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai // *arXiv preprint arXiv:1504.07056.* — 2015.

22. *Wattenhofer, Mirjam.* Distributed weighted matching / Mirjam Wattenhofer, Roger Wattenhofer. — Springer, 2004.

23. *Bertsekas, Dimitri P.* Distributed asynchronous relaxation methods for linear network flow problems / Dimitri P Bertsekas, Jonathan Eckstein. — 1986.

24. *Bartal, Yair.* Probabilistic approximation of metric spaces and its algorithmic applications / Yair Bartal // Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on / IEEE. — 1996. — Pp. 184–193.

25. *Racke, Harald.* Optimal hierarchical decompositions for congestion minimization in networks / Harald Racke // Proceedings of the fortieth annual ACM symposium on Theory of computing / ACM. — 2008. — Pp. 255–264.

26. *Peleg, David.* Distributed computing / David Peleg // *SIAM Monographs on discrete mathematics and applications.* — 2000. — Vol. 5.

27. Executing dynamic data-graph computations deterministically using chromatic scheduling / Tim Kaler, William Hasenplaugh, Tao B Schardl, Charles E Leiserson // Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures / ACM. — 2014. — Pp. 154–165.

28. A distributed algorithm for the maximum flow problem / Thuy Lien Pham, Ivan Lavallee, Marc Bui, Si Hoang Do // Parallel and Distributed Computing, 2005. ISPDC 2005. The 4th International Symposium on / IEEE. — 2005. — Pp. 131–138.

29. Approximate weighted matching on emerging manycore and multithreaded architectures / Mahantesh Halappanavar, Oreste Villa John Feo, Antonino Tumeo, Alex Pothen // *The International Journal of High Performance Computing Applications.* — 2012.

30. *Hoepman, Jaap-Henk.* Simple Distributed Weighted Matchings / Jaap-Henk Hoepman // *arXiv preprint cs/0410047.* — 2008.

31. *Preis, Robert.* Linear Time 1/2 -Approximation Algorithm for Maximum Weighted Matching in General Graphs / Robert Preis // *Springer-Verlag Berlin Heidelberg.* — 1999.

32. *NJ Davis BA Carpenter, CW Glover.* Parallel approaches to the solution of the assignment problem / CW Glover NJ Davis, BA Carpenter // *Concurrency: Practice And Experience.* — 1992.

33. *et al., Thomsen.* Effective caching of shortest paths for location-based services / Thomsen et al. // *SIGMOD.* — 2012. — Pp. 313–324.

34. *Cooper, Keith D.* A simple, fast dominance algorithm / Keith D Cooper, Timothy J Harvey, Ken Kennedy // *Software Practice & Experience.* — 2001. — Vol. 4. — Pp. 1–10.

35. TOUCH: in-memory spatial join by hierarchical data-oriented partitioning / Sadegh Nobari, Farhan Tauheed, Thomas Heinis et al. // Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data / ACM. — 2013. — Pp. 701–712.

36. *Bouros, Panagiotis.* Spatio-textual similarity joins / Panagiotis Bouros, Shen Ge, Nikos Mamoulis // *Proceedings of the VLDB Endowment.* — 2012. — Vol. 6, no. 1. — Pp. 1–12.

37. *Rice, Michael.* Graph indexing of road networks for shortest path queries with label restrictions / Michael Rice, Vassilis J Tsotras // *Proceedings of the VLDB Endowment.* — 2010. — Vol. 4, no. 2. — Pp. 69–80.

38. Towards online shortest path computation / Hong Jun Zhao, Man Lung Yiu, Yuhong Li et al. // *Knowledge and Data Engineering, IEEE Transactions on.* — 2014. — Vol. 26, no. 4. — Pp. 1012–1025.

39. *Thomsen, Jeppe Rishede.* Effective caching of shortest paths for location-based services / Jeppe Rishede Thomsen, Man Lung Yiu, Christian S Jensen // Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data / ACM. — 2012. — Pp. 313–324.

40. *Guttman, Antonin.* R-trees: a dynamic index structure for spatial searching / Antonin Guttman. — ACM, 1984. — Vol. 14.

41. *Greene, Diane.* An implementation and performance analysis of spatial data access methods / Diane Greene // Data Engineering, 1989. Proceedings. Fifth International Conference on / IEEE. — 1989. — Pp. 606–615.

42. The R*-tree: an efficient and robust access method for points and rectangles / Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger. — ACM, 1990. — Vol. 19.

43. *Roussopoulos, Nick.* Nearest neighbor queries / Nick Roussopoulos, Stephen Kelley, Frédéric Vincent // ACM sigmod record / ACM. — Vol. 24. — 1995. — Pp. 71–79.

44. *Hjaltason, Gísli R.* Distance browsing in spatial databases / Gísli R Hjaltason, Hanan Samet // *ACM Transactions on Database Systems (TODS).* — 1999. — Vol. 24, no. 2. — Pp. 265–318.

45. A concurrent k-NN search algorithm for R-tree / Jagat Sesh Challa, Poonam Goyal, S Nikhil et al. // Proceedings of the 8th Annual ACM India Conference / ACM. — 2015. — Pp. 123–128.

46. *Goldberg, Andrew V.* Network Flow Algorithms / Andrew V Goldberg, Eva Tardos, Robert E Tarjan // REPRINT FROM ALGORITHMS AND COMBINATORICS VOLUME 9 / Citeseer.

47. Finding minimum-cost flows by double scaling / Ravindra K Ahuja, Andrew V Goldberg, James B Orlin, Robert E Tarjan // *Mathematical programming.* — 1992. — Vol. 53, no. 1-3. — Pp. 243–266.

48. *Dezső, Balázs.* LEMON–an open source C++ graph template library / Balázs Dezső, Alpár Jüttner, Péter Kovács // *Electronic Notes in Theoretical Computer Science.* — 2011. — Vol. 264, no. 5. — Pp. 23–45.

49. *Shun, Julian.* Ligra: A Lightweight Graph Processing Framework for Shared Memory / Julian Shun, Guy Blelloch // *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 135-146.* — 2013.

50. *Nanongkai, Danupon.* Distributed approximation algorithms for weighted shortest paths / Danupon Nanongkai // Proceedings of the 46th Annual ACM Symposium on Theory of Computing / ACM. — 2014. — Pp. 565–573.

51. Distributed verification and hardness of distributed approximation / Atish Das Sarma, Stephan Holzer, Liah Kor et al. // *SIAM Journal on Computing*. — 2012. — Vol. 41, no. 5. — Pp. 1235–1265.

52. *Dinic, E.A.* Algorithm for solution of a problem of maximum flow in networks with power estimation. / E.A. Dinic // *Soviet Mathematical Docladi 11*. — 1970. — P. 1277–1280.

53. *Mulmuley, Ketan.* A Lower Bound on Computing Blocking Flows in Graphs / Ketan Mulmuley, Pradyut Shah.

54. *Talbi, EG.* Parallel combinatorial optimization / EG Talbi. — Wiley-Interscience, 2006.