International Journal of *Geo-Information*

Article

Solving Spatial Optimization Problems via Lagrangian Relaxation and Automatic Gradient Computation

Zhen Lei [1] **and Ting L. Lei** [2,*]

[1] College of Automation, Wuhan University of Technology, Wuhan 430070, China; leizhen@whut.edu.cn
[2] Department of Geography & Atmospheric Science, University of Kansas, Lawrence, KS 66045, USA
[*] Correspondence: lei@ku.edu

Abstract: Spatial optimization is an integral part of GIS and spatial analysis. It involves making various decisions in space, ranging from the location of public facilities to vehicle routing and political districting. While useful, such problems (especially large problem instances) are often difficult to solve using general mathematical programming (due to their generality). Traditionally, an alternative solution method is Lagrangian relaxation, which, if well-designed, can be fast and optimal. One has to derive the Lagrangian dual problem and its (sub)gradients, and move towards the optimal solution via a search process such as gradient descent. Despite its merits, Lagrangian relaxation as a solution algorithm requires one to derive the (sub)gradients manually, which is error-prone and makes the solution algorithm difficult to develop and highly dependent on the model at hand. This paper aims to ease the development of Lagrangian relaxation algorithms for GIS practitioners by employing the automatic (sub)gradient (autograd) computation capabilities originally developed in modern Deep Learning. Using the classic *p*-median problem as an example, we demonstrate how Lagrangian relaxation can be developed with paper and pencil, and how the (sub)gradient computation derivation can be automated using autograd. As such, the human expert only needs to implement the Lagrangian problem in a scientific computing language (such as Python), and the system can find the (sub)gradients of this code, even if it contains complex loops and conditional statements. We verify that the autograd version of the algorithm is equivalent to the original version with manually derived gradients. By automating the (sub)gradient computation, we significantly lower the cost of developing a Lagrangian algorithm for the *p*-median. And such automation can be applied to numerous other optimization problems.

Keywords: GIS; discrete optimization; Lagrangian relaxation; algorithm; gradient descent

check for updates

Academic Editor: Wolfgang Kainz

Received: 19 October 2024
Revised: 9 December 2024
Accepted: 20 December 2024
Published: 2 January 2025

Citation: Lei, Z.; Lei, T.L. Solving Spatial Optimization Problems via Lagrangian Relaxation and Automatic Gradient Computation. *ISPRS Int. J. Geo-Inf.* **2025**, *14*, 15. https://doi.org/10.3390/ijgi14010015

1. Introduction

Spatial optimization is an integral part of GIS and spatial analysis, and involves making various kinds of decisions in the space optimally. This includes finding the best sites for central facilities such as firefighting stations and hospitals (e.g., using the *p*-median problem), finding the best routes for delivery trucks (e.g., the travelling salesman problem), and districting and zoning problems, among many others. Due to the advancements in computer technologies and operational research since the 1970s, spatial optimization problems in a wide range of domains can now be formulated with relative ease using mathematical programming languages or other algebraic modeling languages. Solving these problems, however, is not as easy due to their inherent complexity. The process of finding an optimal solution can be excessively long, and some may not even converge.

There are at least four ways to solve spatial optimization problems. Firstly, spatial optimization problems, once formulated in Integer Linear Programming (ILP) or other modeling languages, can (often) be directly solved by the corresponding solvers (such as CPLEX or GNU GLPK). This approach is the easiest in terms of algorithmic development, since the optimization model itself *is* the program. However, this solution approach is known to be slow for all except for small problem instances or specific types of problems.

Secondly, one can develop specialized algorithms (e.g., the Dijkstra algorithm for the shortest path problem), and then prove that they will find the optimal solution upon termination. These specialized algorithms are typically fast; however, many spatial optimization problems (such as the classic p-median problem) do not have such direct closed-form algorithms.

Thirdly, one can resort to heuristic algorithms such as interchange heuristics, simulated annealing, and genetic algorithms to find good solutions in a relatively short amount of time. A potential problem with heuristic algorithms is that, generally speaking, they cannot find the optimal solutions. One cannot be sure whether the obtained solution is optimal or know how far the obtained solution is from the optimal solution. This is different from the ILP-based methods in the first category, or the Lagrangian relaxation method studied in this article (both of which can provide proof of optimality).

One last type of solution algorithm is Lagrangian relaxation, the focus of this article. Lagrangian relaxation is based on the formulation of a Lagrangian **dual** problem of the original problem (called the **primal** problem). Typically, it involves a search procedure that successively finds better solutions based on the (sub)gradients of the Lagrangian dual problem. Similar to typical methods for Integer Linear Programming (such as branch and bound), Lagrangian relaxation can provide a proof of optimality. That is, the algorithm knows when an optimal solution is reached. This is because Lagrangian relaxation can provide both a super-optimal (infeasible) solution, called the **dual** solution and a best incumbent feasible solution of the original problem (called the **primal** solution). Their difference is called the "optimality gap". As with the classic solution algorithms for Integer Linear Programming, the Lagrangian-relaxation-based methods reduce this optimality gap as they proceed. Similar to specialized algorithms, a well-designed Lagrangian relaxation algorithm typically converges much faster than an ILP solver.

Despite its merits, a Lagrangian-relaxation-based method can take a significant amount of time to design and implement. As shown in the next section, this is because Lagrangian relaxation requires choosing a Lagrangian function (a.k.a. the relaxed problem) that can be efficiently implemented in a programming language. Furthermore, it requires deriving the mathematical formulae for the subgradients and translating these formulae into a program. Such tasks can be error-prone and difficult, especially for those without prior training in optimization theory. With all this work, there is still no guarantee that the Lagrangian optimization algorithm will converge rapidly. One may still need to step back and try out different ways of relaxation for the original problem. Therefore, the cost of developing a Lagrangian algorithm may be too high in many cases.

The goal of this article is to ease the development of Lagrangian relaxation algorithms by using automatic gradient (autograd) computation that stems from the recent advancements in Deep Learning algorithms. In Deep Learning, gradient computation is routinely used to compute the feedback signals for neural networks. Different auto-gradient packages have been developed to suit such computational needs. We demonstrate that such packages can be utilized to help solve spatial optimization problems by automating the derivation of gradients (of the Lagrangian problems). Once the analyst expresses the solution of the Lagrangian function as a suitable program, the autograd packages can take over from then on, reducing the bulk of the derivation and programming work about the (sub)gradients.

The complexity of the (sub)gradient computation may vary depending on the problem at hand. In this article, we use the classic p-median problem as an example to demonstrate the feasibility of automating such computations. But, as will be discussed later on, with autograd, the same technology can be applied to arbitrarily complex Lagrangian problems provided that they can be implemented in a computer language with conditional (if) and loop statements. To the best of our knowledge, such automation has not been explored in the spatial optimization literature.

It should be noted that the Lagrangian function is not typically differentiable everywhere, which means that the normal gradient in the mathematical sense may not exist. Therefore, the so-called subgradients must be used instead of regular gradients. When there is no ambiguity, we will use the term "gradient computation" and subgradient computation interchangeably hereafter.

In the rest of this paper, we provide a brief review of Lagrangian relaxation and its application to spatial optimization in Section 2. In Section 3, we use the p-median problem as an example to demonstrate how gradients can be computed both manually by paper and pencil and automatically by autograd. In the Section 4, we compare the solutions of the p-median problem obtained by an ILP solver, and by a manual Lagrangian relaxation algorithm, as well as by a Lagrangian relaxation algorithm using autograd. We then conclude with a summary of the findings and suggest possible future work.

2. Background

The Lagrangian relaxation method is based on the use of Lagrangian multipliers, and is widely applicable to constrained optimization problems, including many spatial optimization models. In these problems, the objective that needs to be optimized cannot be easily expressed as a closed-form function; instead, it is expressed implicitly as an objective function plus a set of constraints. In this section, we discuss the main ingredients of the method of Lagrangian multipliers and Lagrangian relaxation with a focus on how to solve spatial optimization problems such as the classic p-median problem [1,2]. Interested readers are referred to the papers and books by [3–5] on a more comprehensive coverage of Lagrangian relaxation and convex optimization.

2.1. Lagrangian Multipliers and Lagrangian Relaxation

2.1.1. Lagrange's Milkmaid Problem

The method of Lagrangian multipliers is best illustrated with the original Lagrange's milkmaid problem (Figure 1). As shown in the figure, a maid is at location o and needs to travel to the destination location d where the cow is and milk the cow. However, there is a constraint that the maid must first walk to the river to wash the bucket. The problem is to find the shortest route from o to d subject to the constraint. It is not difficult to see that the route consists of line segments from the origin o to some point P on the near side of the river bank, and then from P to the destination d.

There are two main forces that determine the optimal solution for the location P in the milkmaid problem (and optimization problems in general): (1) the objective function $f(P)$, and (2) the constraint $g(P) = 0$. The objective function can be expressed as $\min\,(P) = |oP| + |Pd|$. That is, the objective is to minimize the total lengths of the two trips from o to P and P to d. As illustrated in Figure 1, the equidistant lines to points o and d (or isochrones of $f(P)$) are a series of ellipses. The smaller the ellipses, the better the objective value. This is indicated by the gradient vector of the ellipses in the figure (towards the near side).

The second force is the constraint. Without it, the solution will be unconstrained and P will be located on the lowest valued isochrone (i.e., the line od). With the constraint on, the optimal location P is "pulled" away from the unconstrained optima. The farther the river bank is from o and d, the greater the deviation is from the unconstrained optima. This is indicated by the gradient vector of the river bank (towards the far side) in Figure 1. A necessary condition for location P to be optimal is that the two aforementioned gradients cancel each other out, as will be explained shortly.

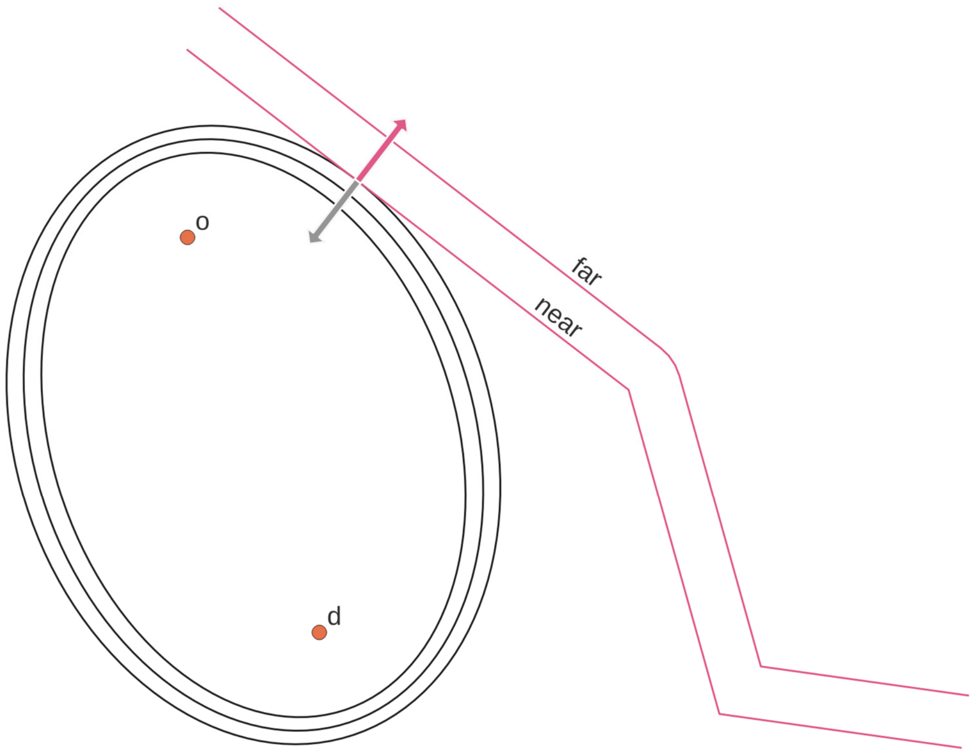


Figure 1. The milkmaid problem. o is the origin, d is the destination.

2.1.2. The Meaning of the Lagrangian Multipliers

For simplicity, we assume that the river bank is a line (we can always do so in an infinitesimally small area). The magnitude of the gradient for the objective function represents the extent to which the objective decreases if the location P is moved by a unit distance along the gradient direction. The magnitude of the constraint gradient represents the extent to which the objective increases if the location P is moved along its direction. These two vectors must be canceled out at the optimal location. Otherwise, there would exist a direction (i.e., the sum of the two gradient vectors projected to the constraint plane) along which the objective can be improved further. Consequently, for any **optimal** solution P^*, we must have the following:

$$f(P^*) - \mu \cdot g(P^*) = 0 \tag{1}$$

Therefore, μ represents the ratio between the magnitudes of the objective gradient and the constraint gradient at optimality. Furthermore, any optimal solution P^* is by definition feasible. Therefore, the original constraint must be satisfied, and we must have the following:

$$g(P^*) = 0 \tag{2}$$

A more compact way of expressing the above two optimality conditions (1) and (2) is to consider the multiplier as a new dimension and define (in a higher dimensional space) the so-called *Lagrangian function* $F(P, \mu)$ of this optimization problem as follows:

$$F(P, \mu) \equiv f(P) - \mu \cdot g(P) \tag{3}$$

Then, the Lagrangian relaxation method optimizes $F(P, \mu)$ instead of the original objective $f(P)$. Now, the original optimality condition can be expressed succinctly in terms of the gradient of the Lagrangian function (3) as follows:

$$\nabla F(P, \mu) = \nabla f(P) - \mu \cdot \nabla g(P) = 0 \tag{4}$$

We can do so because, in terms of partial derivatives, (4) implies, in the original space, the following: $\nabla f(P) - \mu \cdot \nabla g(P) = 0[= \partial F(P, \mu)/\partial P]$ and $g(P) = 0[= \partial F(P, \mu)/\partial \mu]$.

2.1.3. The Sign of the Lagrangian Multiplier

Note that the partial derivative of the Lagrangian function w.r.t. μ is $g(P)$, the left-hand-side (LHS) of the constraint, and should be zero in any feasible solution of the original problem. Therefore, $\partial F(P, \mu)/\partial \mu$ stands for *the amount of violation* of the constraint. After *relaxing* the constraint, the constraint violation generally can be non-zero in a solution of the relaxed problem. In other words, the solution of the relaxed problem is generally infeasible for the original problem. Given the minimization objective here, we know the following:

$$f(P) \geq f(P^*) \geq F(P, \mu) \tag{5}$$

In other words, the solution of the relaxed problem is generally super-optimal (i.e., better than the optimal solution of the original problem). This property is known as weak duality. The difference between the two objective values is known as the *optimality gap*.

In an optimal solution of the relaxed problem, the violation should always be zero, according to (2). Therefore, at optimality, the optimality gap must be zero, and we have the following:

$$f(P) = F(P, \mu) \tag{6}$$

Property (6) is known as the strong duality. We will observe the duality gap and its convergence to zero later on in the Section 4.

When only one constraint is relaxed, its multiplier sign can be easily determined. The main idea is that, in relaxing a constraint, we allow it to be violated, but this violation should be *penalized* in the objective function of the relaxed problem and eventually reduced to zero (at optimality). Therefore, any violation should impact the objective (3) *negatively*. In an equality constraint $g(P) = 0$, the potential violation is two-sided. If the violation at a specific point P is positive, and assuming a minimization objective, we should make μ in (3) negative so that the Lagrangian function becomes worse (greater) with a greater absolute violation. If the violation is negative, we should make μ positive so that the Lagrangian function increases with absolute violation. In short, we can rewrite the Lagrangian function as $(P, \mu) \equiv f(P) - \mu \cdot sgn(g(P)) \cdot |g(P)|$, and see that we must have the following:

$$-\mu \cdot sgn(g(P)) \geq 0 \tag{7}$$

where $sgn(\cdot)$ is the sign function.

When the constraint is in inequality form, the violation is one-sided, and we can pre-determine the sign of the multiplier. For example, given a minimization objective and a less-than-or-equal constraint $g(P) \leq 0$, we must have $\mu \geq 0$ because of (7). When multiple constraints are relaxed, the Lagrangian multiplier becomes a vector. For each component of the vector, the sign rule (7) still holds. We will see in the next section that this knowledge can enable us to reduce the search space of the Lagrangian relaxation.

2.2. Lagrangian Relaxation for Spatial Optimization Problems

With few changes, Lagrangian relaxation can also be used to solve spatial optimization problems formulated in Integer Linear Programming. We use the classic p-median problem [1,2] as an example. The p-median problem can be formulated using Integer Linear Programming [6]. The following notation is needed:

I: customer locations;

a_i: population at $i \in I$;

J: candidate sites for facilities;

d_{ij}: distance from location i to j.

The decision variables for the p-median problems are the assignment variable x_{ij} and location variable y_j:

$$x_{ij} = 1, \text{if } i \text{ is assigned to } j \in J, \text{ or 0 otherwise} \tag{8}$$

$$y_j = 1, \text{if a facility is at } j \in J, \text{or 0 otherwise} \tag{9}$$

Then, the p-median problem is to

$$\text{minimize} \sum_{i \in I, j \in J} a_i \cdot d_{ij} \cdot x_{ij} \tag{10}$$

subject to

$$\sum_{j \in J} x_{ij} = 1, \forall i \in I \tag{11}$$

$$\sum_{j \in J} y_j \leq p \tag{12}$$

$$x_{ij} \leq y_j, \forall i \in I, j \in J \tag{13}$$

Objective function (10) minimizes the total population weighted service distance. Constraint (11) maintains that each customer i should be assigned to one facility. Constraint (12) maintains that, at most, p candidate sites can be selected as facilities. Constraint (13) refers to the Balinski constraints that relate to the assignment and location variables by stating that a customer can only be assigned to an open facility.

In the milkmaid problem, we moved all the constraints (the only constraint) into the objective function. With more complex problems such as the p-median, one can move a *subset* of the constraints into the objective, and leave other constraints as they are. The constraints that are moved into the objective Lagrangian function are said to have been **relaxed** or **dualized**. Conceptually, the remaining constraints and the modified objective function form an implicit Lagrangian function. We optimize it in a higher-dimensional space with the Lagrangian multipliers. This general process of relaxing *some* of the constraints is called Lagrangian relaxation.

For the p-median problem, the cardinality constraint (12) can be relaxed. In the relaxed problem, demands can be assigned to more than p facilities. But the over-location (opening more than p facilities) is penalized (with a multiplier λ):

$$\text{minimize} \sum_{i \in I, j \in J} a_i \cdot c_{ij} \cdot x_{ij} + \lambda \cdot \left(\sum_{j \in J} y_j - p \right) \tag{14}$$

s.t. (12) (13)

The relaxed problem has the structure of a Simple Plant Location Problem (SPLP), with multiplier λ being the fixed cost in SPLP. The relaxed problem for each λ can be solved using a solution algorithm for SPLP such as the DUALOC algorithm. Since there is only one Lagrangian multiplier, one can use various interval reduction methods (such as bisection and linear interpolation) [7] to determine the optimal λ value.

Alternatively, we can relax the assignment constraint (11) to obtain a new relaxed problem. This is the approach taken by [4], and the approach that we follow in this article. Since many multipliers are required (one μ_i for each relaxed constraint (11), the interval search method ceases to work.

In this higher-dimensional space of multipliers, a general method for determining the optimal multiplier values is the gradient descent method. Conceptually, the Lagrangian dual problem can be solved on two levels. On the first (lower) level, we assume that the multiplier values μ_i are fixed and optimize the (implicit) Lagrangian function with respect to the original/primal decision variables (x_{ij}, y_j). This problem is called the **relaxed problem** or **Lagrangian function** from now on. It produces primal variable values as functions of (temporarily fixed) multiplier values.

On the second (higher) level, we solve the higher level Lagrangian dual problem by allowing the multiplier value to vary and applying an optimization technique to the multipliers to find a series of better solutions until an optimal solution is reached. In this paper, we perform a simple gradient descent search while optimizing for the multipliers, as described in the next section.

2.3. Computational Graphs and Automatic Gradient Computation (Autograd)

The gradients of the Lagrangian dual problem are conventionally computed by theoretical deviation with paper and pencil. This is not only cumbersome and error-prone, but also requires handling special cases in which gradients in the normal sense do not exist. Fortunately, the gradients for many optimization problems can be computed automatically and symbolically by constructing so-called computational graphs. Mathematically, a function of a set of input data can be represented as a set of connected edges leading from one data item to another, which eventually leads to the final function value. In our context, the final function value is the Lagrangian function value. In this graph, each node is a data item, which is often represented as a multi-dimensional array (or a tensor). Each edge represents a computational step leading from this tensor to the result of that step. Naturally, these edges form a directed acyclic graph (DAG) with the input data as the leaves and the final objective function value as the root node. In the Deep Learning literature, this formation process is often called the **forward pass**, referring to the fact that one starts from the input and successively constructs intermediate values at each node until the final value root node is computed.

Once constructed, the computational graph contains a full history of how the final objective value is obtained. The history is recorded as a hierarchy of small functions at each edge, which, when composed, expresses a big function that maps the input data to the objective value. Using the *chain rule* in calculus, one can then express the gradient of the large function as the gradients of the small functions. By the chain rule, the gradient computation goes backward starting from the root node value and stopping at the leaf nodes. This is why the gradient computation following the computational graph is referred to as the **backward pass**.

There are two approaches with which to construct computational graphs: static and dynamic. In static construction, the analyst directly defines the computational graph in a specialized (graph) language and uses it to run the data through the graph. The advantage of static computational graphs is that they allow certain optimizations of the graphs. In dynamic construction, a computational graph is built implicitly by the computer during the forward pass. Since the computational graph is a faithful representation of the computational steps of the objective function, it is compiled immediately after the objective function value is computed. One advantage of dynamic computational graphs is their ease of implementation because their construction does not require a separate language. The optimization problem can be written in a familiar language, such as Python, and the associated computational graph is computed behind the scene. Moreover, the dynamic construction of computational graphs allows flow control statements, such as loops and conditional statements in Python, to be used directly in writing the optimization function. This means that complex optimization functions can be implemented using constructs of conventional programming languages without worrying about how their gradients should be computed.

Owing to the aforementioned advantages, we adopt dynamic computational graphs as implemented in the Python autograd library [8]. Compared to conventional methods, the analyst is freed from the theoretical work on deriving the formula of gradients of their programs, and delegates that task to the computer. Instead, the analyst can focus on developing the optimization code (in the forward pass) in a scientific computing language such as Python or Julia.

3. Methodology

3.1. Formulation

In this section, we present the main Lagrangian relaxation algorithm for the p-median problem. In our algorithm, we relax the assignment constraints (11). This is the same set of constraints as used by Hanjoul and Peeters [4] (in their "Relaxation 2" method). However, there are a number of differences. Firstly, ref. [4]'s method involved a tree search process, which starts from the primal problem space, and its success depends on a good initial primal solution (obtained from an interchange heuristic). In contrast, we adopt a gradient optimization approach that works in the dual problem space and does not require an initial primal solution. Instead, we start from an initial dual solution and move successively towards an optimal solution. Primal solutions are only generated to measure the duality gap and compute (sub)gradients in the dual space. Secondly, we modify constraint (11) into an inequality form to reduce the search space of the multipliers, as will be explained briefly. In our experiments, we found that the performance of our LR algorithm was satisfactory. The (modified) p-median formulation is as follows:

$$\text{minimize} \sum_{i\in I, j\in J} a_i \cdot d_{ij} \cdot x_{ij} \tag{15}$$

subject to

$$1 - \sum_{j\in J} x_{ij} \leq 0, \forall i \in I \tag{16}$$

$$\sum_{j\in J} y_j \leq p \tag{17}$$

$$x_{ij} \leq y_j, \forall i \in I, j \in J \tag{18}$$

Constraint (11) is transformed into an equivalent inequality form (16). In the context of the p-median problem, the new form (16) allows one customer to be assigned to more than one server. But, at optimality, one customer will be assigned only to one server, because assigning the customer to additional servers will only increase the objective function. Therefore, the new form (16) is equivalent to the original form (11). The reason we use the modified constraint (16) is that, according to the sign rule (7) in the previous section, the Lagrangian multiplier for such an inequality constraint must be non-negative. Therefore, we reduce the search space for the Lagrangian problem by one half, which accelerates the gradient descent process. The rest of the model is the same as the original p-median formulation. With this in mind, the Lagrangian relaxed problem (i.e., Lagrangian function) is as follows:

$$\text{minimize} \sum_{i\in I, j\in J} a_i \cdot d_{ij} \cdot x_{ij} + \sum_i \mu_i \cdot \left(1 - \sum_{j\in J} x_{ij}\right) \tag{19}$$

subject to (17), (18), and

$$\mu_i \geq 0, \quad \forall i \in I \tag{20}$$

For the Lagrangian function, the values of the multipliers μ_i are fixed. The objective function (19) can be re-arranged as follows:

$$\text{minimize} \sum_{i\in I} \mu_i - \sum_{j\in J} \left(\sum_{i\in I} (\mu_i - a_i \cdot d_{ij}) \cdot x_{ij} \right) \tag{21}$$

In the above equation, the first term is a constant. In the second term, by constraints (17) and (18), at most, p facilities can be open, and only for the p corresponding to j values can we have $x_{ij} = 1$. For the other j values, the corresponding facilities are closed, and we must have $x_{ij} = 0$. This means that only p of the entries in $\sum_{i\in I} (\mu_i - a_i \cdot d_{ij}) \cdot x_{ij},\ i \in I$ can be non-zero.

As pointed out in [3,4], at optimality of the Lagrangian problem, x_{ij} is determined by the y_j value. If the coefficient $\mu_i - a_i \cdot d_{ij}$ is positive, then x_{ij} should be set to its upper bound, namely, y_j (by (18)); otherwise, x_{ij} should be set to its lower bound, 0. That is, at optimality, we set

$$x^*_{ij} = \begin{cases} y^*_j, & \text{if } \mu_i - a_i \cdot d_{ij} > 0 \\ 0, & \text{otherwise} \end{cases} \tag{22}$$

where x^*_{ij}, and y^*_j are the optimal values for the Lagrangian problem.

Using the relationship between x_{ij} and y_j, we can rewrite the Lagrangian function's objective again as follows:

$$\text{minimize} \sum_{i\in I} \mu_i - \sum_{j\in J} \left(\sum_{i\in I} (\mu_i - a_i \cdot d_{ij})^+ \right) \cdot y_j \tag{23}$$

where $(x)^+ := max(x, 0)$.

Because the only effective constraint is now the cardinality constraint (17), we can sort the coefficients of y_j, pick the largest p entries, and set the corresponding y_js to one. Algorithmically, this means that, to compute the Lagrangian function, one simply needs to sort the array of y_j coefficients, add up the largest p entries and subtract them from the sum of multipliers.

Figure 2 presents the overall workflow of the Lagrangian optimization algorithm. In essence, the Lagrangian relaxation algorithm is a bi-level optimization problem. For the (lower-level) optimization described in this subsection, we fix the values of the multipliers μ_i, and optimize (minimize) the objective (23) w.r.t. the primal variable y_j. This solves the Lagrangian function (the lower-level problem with fixed multiplier values). We convert this dual solution into a primal solution by simply keeping the values of the y_j variables and then deriving the x_{ij} values by the closest assignment of customers to facilities. The difference between the primal and dual solution is the optimality gap with which we determine whether the algorithm should be stopped.

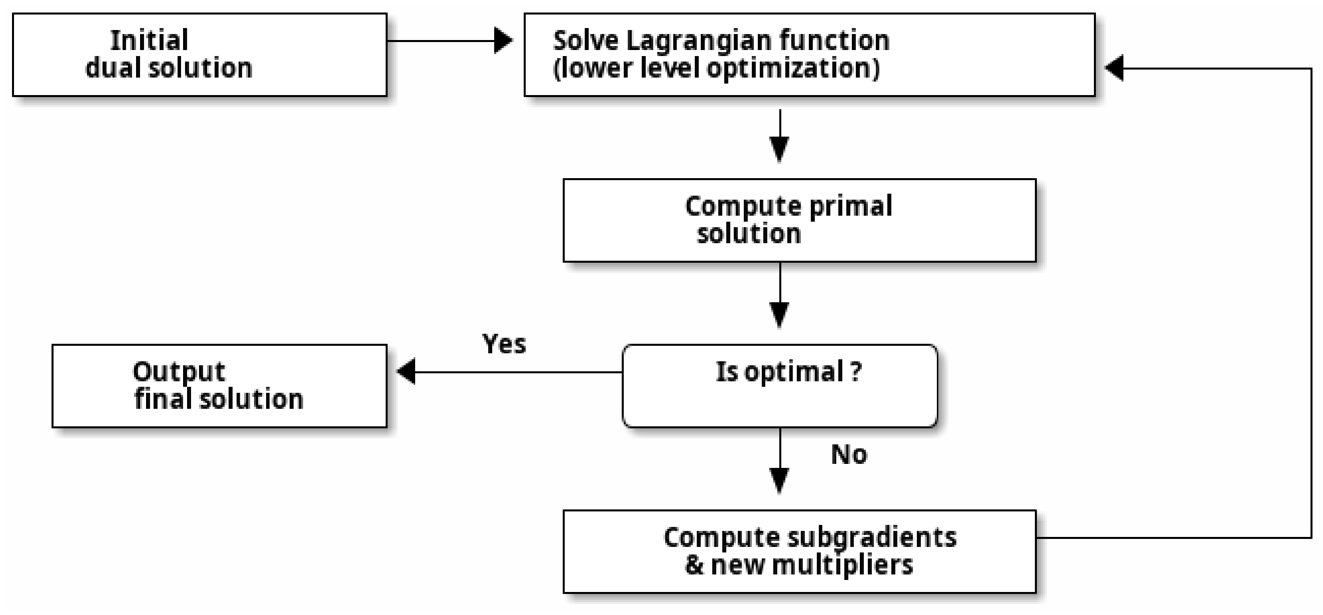

Initial dual solution
Solve Lagrangian function (lower level optimization)
Compute primal solution
Is optimal ?
Yes
Output final solution
No
Compute subgradients & new multipliers

Figure 2. Workflow for the Lagrangian relaxation algorithm using gradient optimization.

After solving the Lagrangian function, we will allow the multipliers to vary in a higher-level optimization problem (i.e., the Lagrangian dual problem) in search of improved dual solutions. In particular, we will optimize the optimal solution of the lower-level Lagrangian function w.r.t. the multipliers (i.e., the dual variables μ_i) by computing the gradients of the Lagrangian function. Based on these gradients, we can compute a set of new multiplier values (as will be discussed shortly) giving rise to a new iteration of the search. In the flowchart in Figure 2, solving Lagrangian function constitutes the lower-level optimization (optimizing primal variables) and the overall loop structure constitutes the higher-level optimization problem (optimizing the dual variables/multipliers).

3.2. Subgradient Computation

Taking the derivative of objective function (19) with respect to the multipliers μ_i's, we can obtain the gradient vector θ at the values x^*_{ij}, y^*_j of the Lagrangian problem as follows:

$$\theta_i = 1 - \sum_{j \in J} x^*_{ij} \tag{24}$$

Let $q_i = \sum_{j \in J} x^*_{ij}$; then, $\theta_i = 1 - q_i$. For each customer i, we inspect the p chosen facilities, and compute the q_i (and, therefore, θ_i) as follows: We loop through i. By (22), we only need to inspect the js for open facilities (for which $y^*_j = 1$). For each open facility j, if $\mu_i - a_i \cdot d_{ij} > 0$, we add one to the counter. In the end, the counter value is q_i. It is not difficult to translate the above logic into code.

Now that we have the gradient direction, one remaining issue is to decide how far we want to move the multipliers along this direction. A commonly used formula [4] is to compute the step size as follows:

$$s = \frac{\tau \cdot \left(\widehat{Z_P} - Z_D\right)}{|\theta|^2} \tag{25}$$

where $\widehat{Z_P}$ is the best known primal objective value, and Z_D is the current dual objective value. τ is a parameter in the range of $(0, 2)$. Operation-wise, τ can be viewed as the learning rate parameter used in Deep Learning in that it controls the speed of the gradient descent. However, it has a known range between 0 and 2 from the literature [4]. Based on this range and empirical tests, we use an initial τ value of 0.5. And we reduce the τ by one half after every 200 steps until τ is less than 0.01.

Once the gradient is computed, it can be used to modify the multipliers. Care must be taken to adjust the new multiplier values to be non-negative according to (20). Considering this, the new multiplier vector is as follows:

$$\mu' = \mu + s \cdot \theta \tag{26}$$

It should be noted that the gradient with respect to μ_i does not always exist. For example, in objective function (23), there is a non-smooth term $\left(\mu_i - a_i \cdot d_{ij}\right)^+$, where $(x)^+ := max(x, 0)$ denotes the ReLU activation function in the NN literature. This term is not smooth around the point $\mu_i = a_i \cdot d_{ij}$ because the derivative w.r.t. μ_i is one when μ_i approaches $a_i \cdot d_{ij}$ from the right, and zero when approaching $a_i \cdot d_{ij}$ from the left. Therefore, two different gradient values are possible. In this case, the entire range between the two values (i.e., [0, 1]) is called the sub-differential of $\left(\mu_i - a_i \cdot d_{ij}\right)^+$ at the point $\mu_i = a_i \cdot d_{ij}$. **Any** value in the sub-differential is called a subgradient at this point. Considering all the $|I|$ dimensions together, the sub-differential at $\mu = \left(a_1 d_{1j}, a_2 d_{2j}, ..., a_m d_{mj}\right)$ is a much larger set with many different subgradient vectors.

We can see that the subgradient is a generalization of gradients in the normal sense. By the principles of the subgradient descent, any subgradient vector in the sub-differential can be used. In this case, we arbitrarily pick the gradient of $\left(\mu_i - a_i \cdot d_{ij}\right)^+$ at $a_i d_{ij}$ to be zero.

In general, constructs such as the ReLU function or the absolute value function are typically implemented in programming using a conditional statement or its equivalents (e.g., *if x > 0, then... else...*). Such statements can introduce discontinuity or non-smoothness into the gradient computation, and such anomalies need to be handled appropriately in the subgradient optimization.

3.3. Automatic Gradient Computation (Autograd)

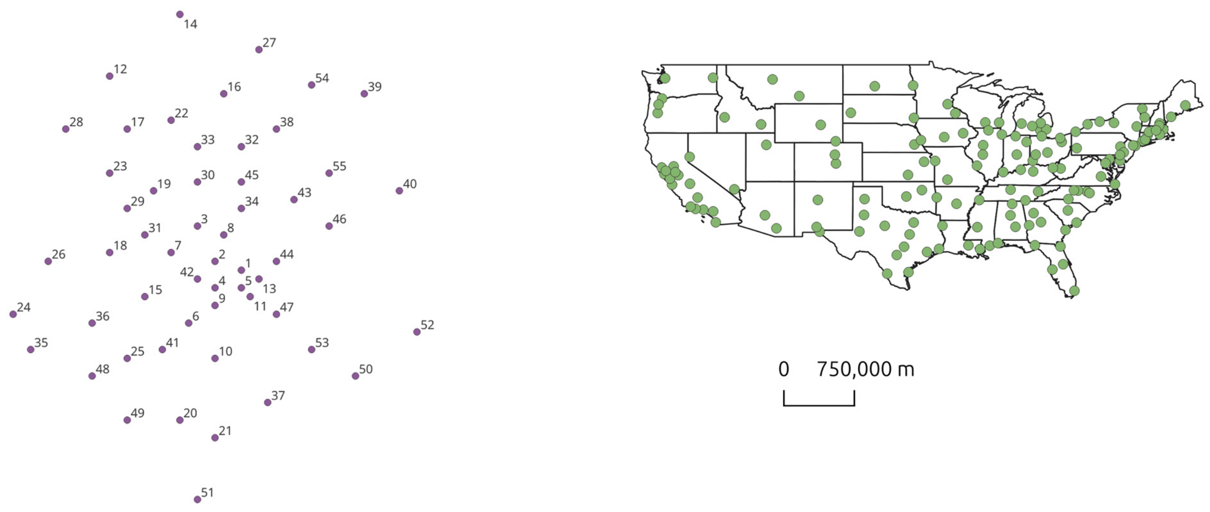
As mentioned in the Section 2, we use the dynamic computational graphs to automatically compute the gradients. In our implementation, we employ the Python autograd library, which makes the gradient computation fully automated. Given a Python function *calc_zdual()* in our code (for computing the objective value of the Lagrangian function), all that is needed is to call the *grad()* function on *calc_zdual*. The return value of *grad(calc_zdual)* is itself a function, which takes the same number and types of parameters as *calc_zdual* but computes its gradient instead of function value. This significantly reduces the complexity of the code and, incidentally, improves its readability. Additionally, autograd has the capability of computing subgradients built in. By using autograd, we avoid the possibility of making errors in gradient computation and take care of the anomalies such as the aforementioned non-smoothness.

It should be noted that autograd can only be used to generate the (sub)gradients. It is still up to the analyst to appropriately use the generated subgradients. In our case, the multipliers μ_i must be adjusted to be non-negative after applying the gradient update in (26). This is achieved by setting $\mu' := max(\mu', 0)$.

4. Experiments

In this section, we present the experimental results of the Lagrangian relaxation algorithm in terms of its convergence behavior and solution characteristics. The computational experiments were executed on a machine with an Intel i9-13900K CPU and 96 Gigabyte of system memory. For comparison purposes, we implemented two versions of the Lagrangian algorithm: one with explicitly derived gradients, and the other with gradients that were automatically derived using the Python autograd package [8]. We also implemented an Integer Linear Programming (ILP) model of the *p*-median problem (19) through (20) using the RElational Linear Programming (RELP) framework [9].

Two datasets are used to test the *p*-median and other central facility location problems. The first is the Swain [10] 55 node dataset (called swain55 hereafter), which is widely used in the spatial optimization literature. As with the literature, the Swain dataset is used both as the sites for customers and as the sites for candidate facilities. Figure 3a presents the layout of the 55 facilities, each representing the centroid of a postcode zone in Baltimore, MD, USA. Each site is labeled with its ID number. The ID numbers are assigned according to the rank of the population, so that site 1 has the highest population, site 2 has the second highest population, and so on. The associated population value (weight) for each site is listed in the "Pop" column in Table 1. From Figure 3 and Table 1, we can observe that the population is approximately concentrated in the center of the region.



(**a**) the Swain 55 dataset (**b**) the largest 150 US cities

Figure 3. Test datasets: Swain 55 node and US Cities 150 node datasets.

Table 1. The Swain 55 node data. The weight values represent the population at each customer location.

ID	Pop.	ID	Pop.	ID	Pop.	ID	Pop.
1	71	15	12	29	6	43	4
2	62	16	11	30	6	44	4
3	56	17	10	31	6	45	3
4	39	18	10	32	5	46	3
5	35	19	9	33	5	47	3
6	21	20	9	34	5	48	3
7	20	21	9	35	5	49	3
8	19	22	8	36	5	50	3
9	17	23	8	37	5	51	3
10	17	24	8	38	4	52	2
11	16	25	8	39	4	53	2
12	15	26	7	40	4	54	2
13	14	27	6	41	4	55	2
14	12	28	6	42	4		

For comparison purposes, we also use the US Cities 150 dataset [11] in Figure 3b as a second test dataset (called cities150 hereafter). It consists of the largest 150 cities in the United States. St. Paul and Minneapolis are treated as one city. The population is based on the 1990 US census and rounded to the nearest 10,000. And distances between cities are computed from the latitudes and longitudes from [11] using the geodesic distance. This dataset is very different from the swain55 dataset in the spatial distribution of the population, as the US population is more concentrated in the east and west coasts rather than in the central regions.

4.1. Initial Solution

As mentioned in the Section 3, our Lagrangian relaxation algorithm primarily works in the dual problem space. We do not assume knowledge about the solution of the Lagrangian dual problem except that the multipliers should be non-negative. Therefore, we arbitrarily choose an initial multiplier value of $\mu_i = 1.0$ for each customer $i \in I$, which leads to a usable dual solution after solving the relaxed problem (Lagrangian function) under these multiplier values. This means that each potential violation of constraint (16) is assigned an equal penalty. And the algorithm will then adjust the multiplier values adaptively by means of the subgradient optimization. Predictably, for $p = 5$ in the Swain 55 node dataset, the initial dual objective is 50 (there are $55 - 5 = 50$ violations, each being 1.0 in worth).

The initial solution of the Lagrangian dual problem contains a set of facility selections y_j, which can be used to construct a feasible solution in the primal domain because we kept the cardinality constraint (17) in the relaxed problem and exactly p facilities are in the dual solution. We only need to set the values of the variable x_{ij} appropriately by assigning each customer to the closest open facility (those with $y_j = 1$). For the problem instance at $p = 5$, the above process leads to an initial primal solution with the facility set [1, 30, 31, 32, 33] and an associated primal objective value of 3966.86. Given the initial dual objective of 50, the optimality gap is computed as follows:

$$gap = \frac{\widehat{Z_P} - \widehat{Z_D}}{Z_P}$$

The initial percentage optimality gap is 98.73%, indicating that the optimal objective can be 98.73% smaller than the incumbent primal solution seen thus far (3966.86). This means that the incumbent solution can be a very poor solution, as the optimal solution's objective value can be anywhere in the whole range of 50 and almost 4000.

4.2. Convergence

The optimality gap is used as the performance metric for our Lagrangian optimization algorithm. Obviously, if the optimality gap is decreased to zero, the algorithm succeeds in finding the optimal solution. Figure 4 depicts the optimality gap as the algorithm proceeds (for the swain55 and cities150 datasets, respectively). In the figure, we present the best primal objective value (the incumbent primal objective) as well as the dual objective values at each iteration of the algorithm. Their difference is the absolute optimality gap. The percent optimality gap is defined as the ratio between the absolute optimality gap and the incumbent primal objective value. Additionally, we plot the curves for both versions of the Lagrangian-based algorithm: one with explicitly derived gradients and the other with automatically computed gradients.

The general trend in Figure 4a demonstrates the essential features of our Lagrangian relaxation algorithm. Basically, the subgradient optimization works primarily in the dual problem space. Starting from the initial solution ($\mu_i = 1.0$ for all i), the algorithm moves successively to a better solution in the dual space guided by the gradient direction θ and the stepsize (25). Correspondingly, we can observe a gradual increase in the dual objective value in Figure 4a. What is interesting is that the primal solution derived from the dual solution (by discarding the x_{ij} variables) is often improved as well, even though we do not explicitly optimize in the primal domain. This is not surprising because, when the dual solution improves, the values of its decision variables become closer and closer to an optimal solution, although the dual solutions are still "super-optimal" (infeasible w.r.t. to the primal problem). Consequently, it is likely that the location variables y_js become closer and closer to an optimal solution as well (since the y_j values as a subset of the dual solution are also feasible in the primal domain). From a different perspective, the dual problem can be viewed as an approximate version of the primal problem where some constraints become "soft" constraints. When one improves the approximate solution, it is natural that some ingredients of it constitute an improved solution for the original/primal problem. Note that the derived primal solutions do not necessarily improve at every iteration. And we only record the best primal objective value observed at each iteration (the incumbent value). This incumbent value is the only thing we need from the primal problem domain in order to determine how far the solution process is from reaching optimality (i.e., the optimality gap). This is why the primal objective curves in Figure 4 descend in a step-wise manner.

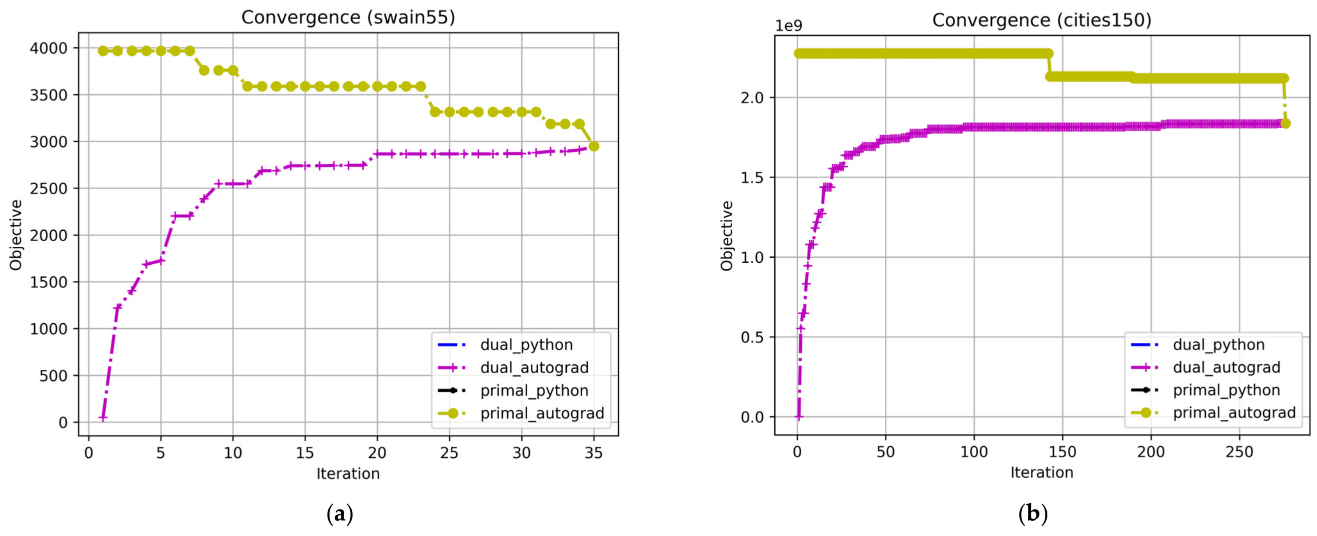


Figure 4. The performance of solvers: the # iteration vs. the primal and dual objectives.

Generally, Figure 4a shows that the optimality gap narrowed down as the number of iterations increased. The primal objective decreased and the dual objective increased until they eventually "meet", at which point the algorithm terminates. It is noteworthy that the optimal objective value must lie somewhere within this narrowing gap. Mathematically, this is guaranteed by the weak duality of Lagrangian relaxation. We stop the algorithm if the optimality gap is below half a percent (0.5%). This means that the optimal solution cannot be different from what the algorithm finds by half a percent. From Figure 3a,b, we can observe that the distance between the dual and the primal objective curve become indiscernible in the end. For most applications, this threshold is sufficient. Of course, one can set a lower threshold value for the optimality gap (at the cost of a longer computation time).

Note that, in Figure 3a, the primal objective curves for the primitive version and the autograd version of the Lagrangian algorithm coincide, and so do the dual objective curves. This indicates that the autograd process has correctly computed the gradients and solutions, which are identical to the primitive version.

Figure 4b demonstrates a similar converging trend of the dual and primal problem for the larger cities150 dataset except that the objective values are much larger (as the between city distance is measured in meters). The number of steps to convergence is also much higher. While the swain55 dataset took 35 steps to converge, the cities150 dataset took 276 steps. This is because the p-median problem is NP-hard. And the computational cost increases very rapidly with the increase in data size (as is the case with many combinatorial optimization problems).

4.3. Solution Characteristics

Figure 5 presents the optimal objective values for various values, as reported by the Lagrangian algorithm (autograd version) and the MILP solver, respectively. The value of p ranged from 3 to 25 at a step size of two. First of all, we can observe that the objective curves for the Lagrangian algorithm and the MILP model coincide. This indicates that, for each value of p, the Lagrangian algorithm found the correct optimal solution.

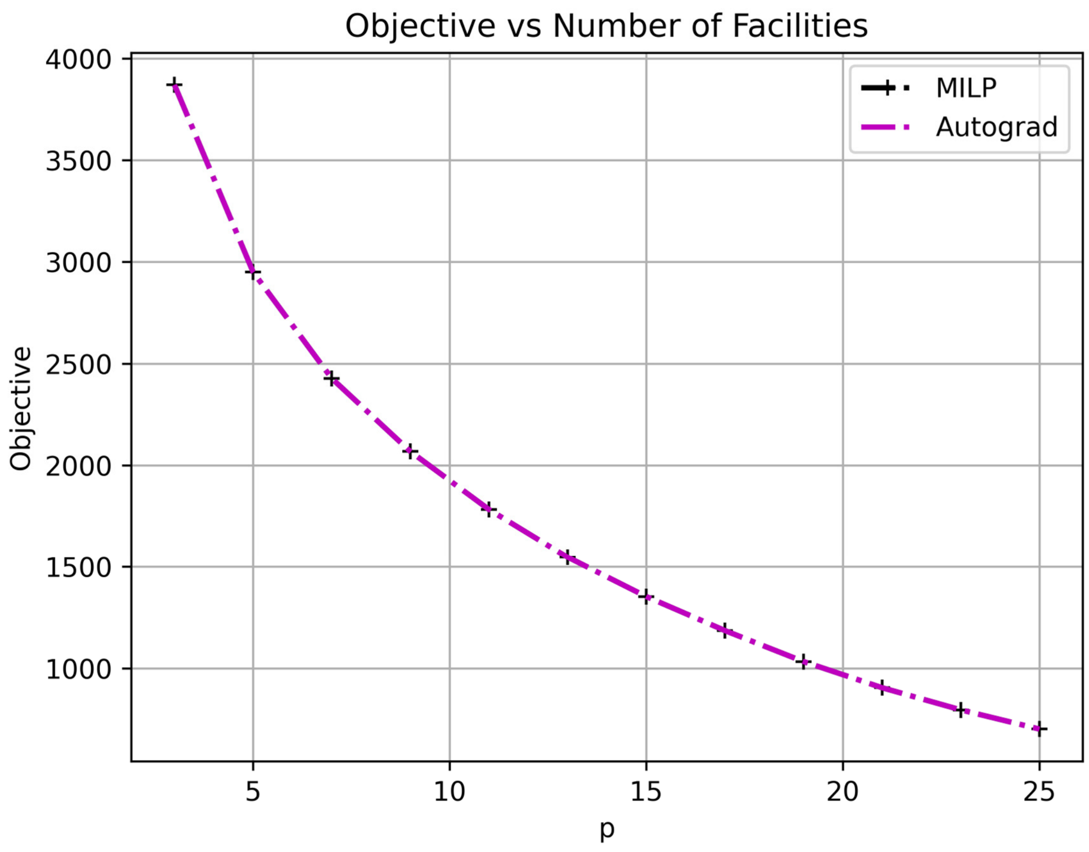


Figure 5. Solution characteristics: the value of p vs. the objective.

Secondly, we can observe a decreasing trend in the optimal p-median objective value as p increases. At $p = 3$, the optimal objective is 3870.24, whereas, at $p = 25$, the optimal objective is only 702.77. This makes sense because, when we increase the number of open facilities, each customer has a greater chance of finding a nearby facility for service. When $p = 25$, nearly half of the candidate sites are chosen to be open facilities, and, therefore, have zero service distance. The service distance for the rest should also be small given a one-to-two facility–customer ratio.

4.4. Computational Cost

Figure 6 presents the number of iterations it took the algorithm to converge at each p for the two sets of test data. From the Figure 6a, the two versions of Lagrangian algorithms have exactly the same number of iterations at each p for the swain55 dataset. This suggests that the manually computed gradients are exactly the same as the autograd computed values, which verifies the correctness of the autograd-based method again. Figure 6b shows the same result for the cities150 dataset.

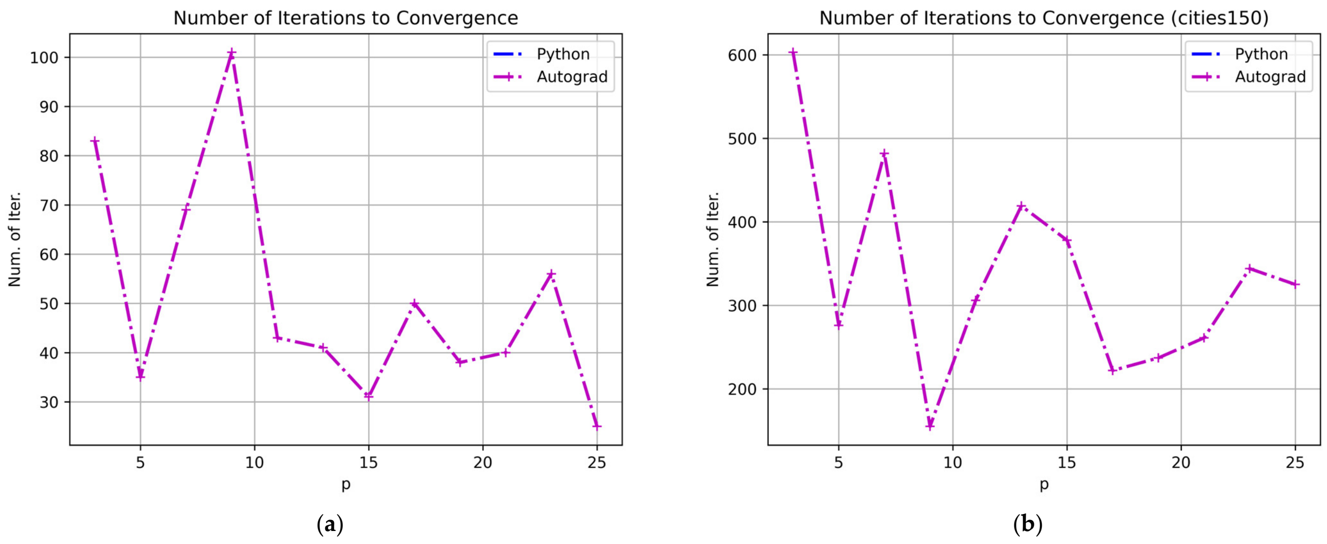


Figure 6. The total number of iterations at each value of p.

Figure 7 presents the computational times for the two Lagrangian algorithms and the MILP solver for the two datasets. For the smaller swain55 dataset, significant fluctuations in the computational time can be observed in Figure 7a. There is no clear conclusion as to which algorithm is faster. But the solution times are generally small. On average, the original, the autograd, and the MILP algorithms took 0.09, 0.10, and 0.12 s, respectively. These fluctuations may be caused by the random delays in loading data between the disk, the memory, and the CPU. For the larger cities150 dataset, Figure 7b shows that the computational time for all three methods are longer and there is a clear pattern, with the original algorithm taking the shortest time, the autograd taking a slightly longer time, and the MILP algorithm being the slowest. On average, the original, the autograd, and the MILP algorithms took 0.27, 0.35, and 2.1 s, respectively.

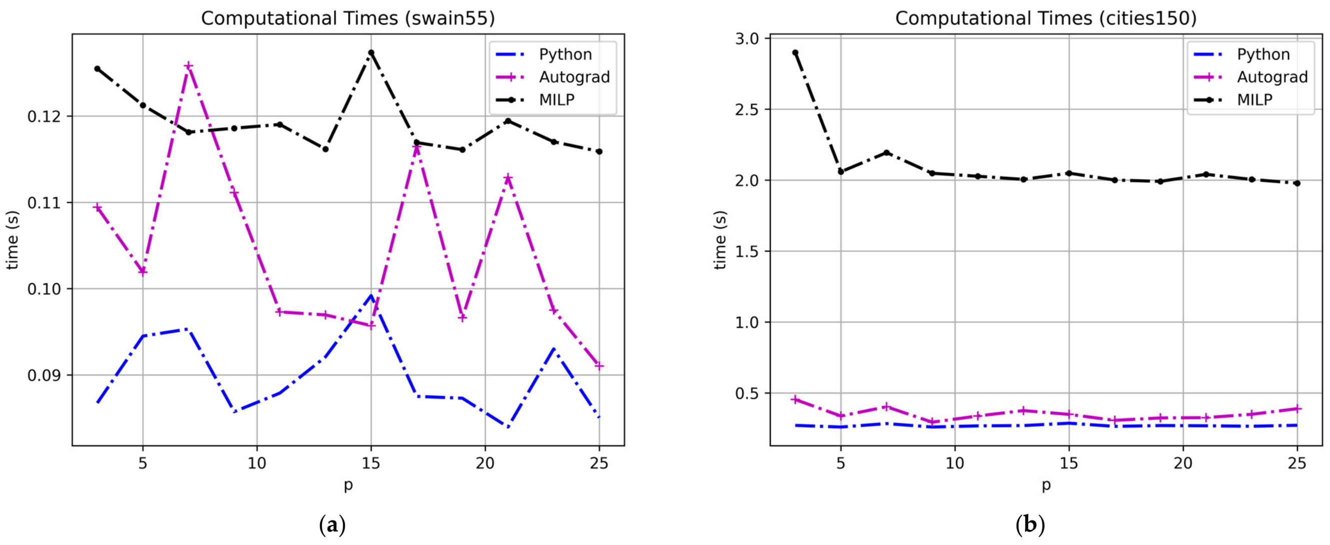
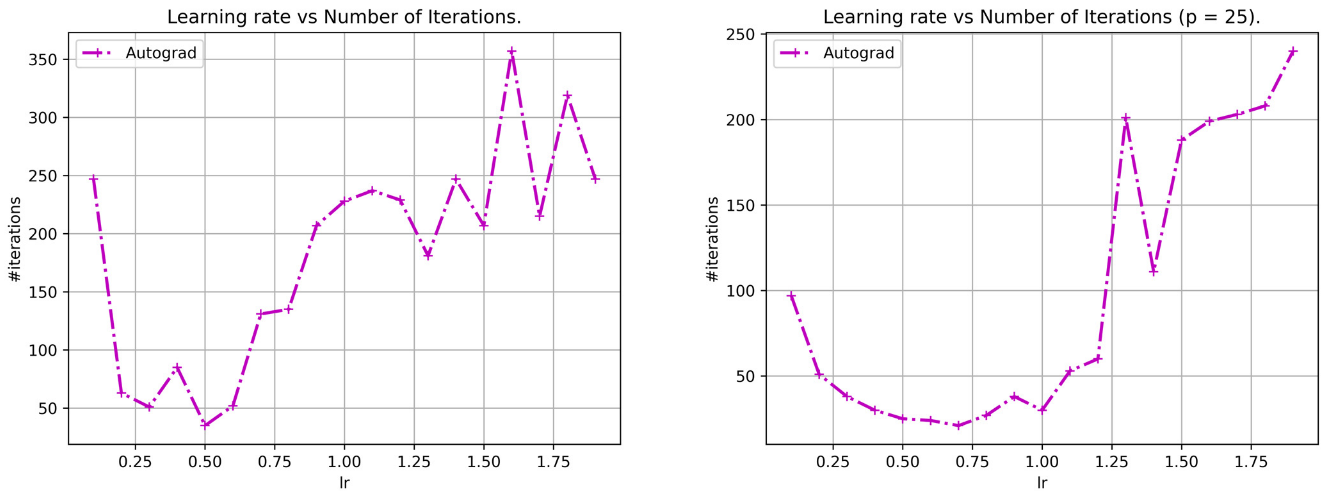


Figure 7. The p value vs. computational time.

We believe that the higher computational times of the autograd-based Lagrangian algorithm are associated with the autograd process itself. As mentioned in the Section 2, the autograd library needs to build an additional computational graph for the Lagrangian dual problem. Therefore, automatic gradient computation comes with a (modest) cost. However, autograd dispenses with the need for deriving the gradient formulae and implementing them in a programming language, and, therefore, reduces a large part of the algorithm development time. For the GIS analyst without prior training in optimization, the work in deriving gradients may even be a barrier, and autograd removes this barrier.

4.5. Sensitivity

Just as with Deep Learning algorithms, an important parameter for our Lagrangian relaxation algorithm is the "learning rate" τ. However, in our case, the range of τ is known to be between 0.0 and 2.0, as mentioned in the Section 3. Thus far, we have used a default τ value of 0.5 in the experiments. To test the sensitivity of the algorithm w.r.t. the learning rate, we have also tested τ values ranging from 0.1 to 1.9 at a 0.1 step size. A key performance metric is the number of steps to convergence, since other metrics such as the running time depend on it. Therefore, we present the number of iterations to convergence in Figures 7 and 8 for the swain55 and cities150 datasets, respectively.



(**a**) (**b**)

Figure 8. Learning rate vs. the number of iterations.

Figure 8 presents the sensitivity results for the swain55 dataset for p = 5 (Figure 8a) and p = 25 (Figure 8b), respectively. We can observe that, for p = 5, the number of iterations range from about 40 to 360, with the smallest number of iterations at $\tau = 0.3$ and $\tau = 0.5$. For p = 25, the number of iterations range from about 20 to 240, with lower numbers achieved when τ is in the 0.2 to 0.8 range.

Figure 9 presents the sensitivity of τ for the larger cities150 dataset. For p = 5 (Figure 9a) and p = 25 (Figure 9b), the number of iterations to termination ranges from about 190 to 830, and from about 210 to 710, respectively. Once more, the algorithm converges faster when the learning rate is within the 0.2 to 0.8 range. It seems that the learning rate should not be too small or too large. Otherwise, the number of steps to convergence can increase significantly.

Figure 10 presents the running time for the swain55 and cities150 datasets (p = 5), respectively. We can observe that the running time ranges from about 0.09 to 0.17 s for the smaller swain55 dataset and ranges from about 0.3 to 0.5 s for the larger cities150 dataset. The general trend of the running time curves approximately follows the curves of the number of iterations, as shown in Figure 8. And the running times are shorter when the learning rate is not too large or too small (between 0.2 and 0.8).

Overall, the computational experiments show that the automatic gradient computation is correct and that all its outcomes are consistent with the hand-crafted gradient formulae. The primal solutions, dual solutions, and paths to convergence are exactly the same. The Lagrangian algorithms generate the same optimal solutions as Integer-Linear-Programming-based solvers. The automatic gradient computation did cause a (modest) increase in computational time. The computational time for the autograd algorithm is generally within one second for the two test datasets.

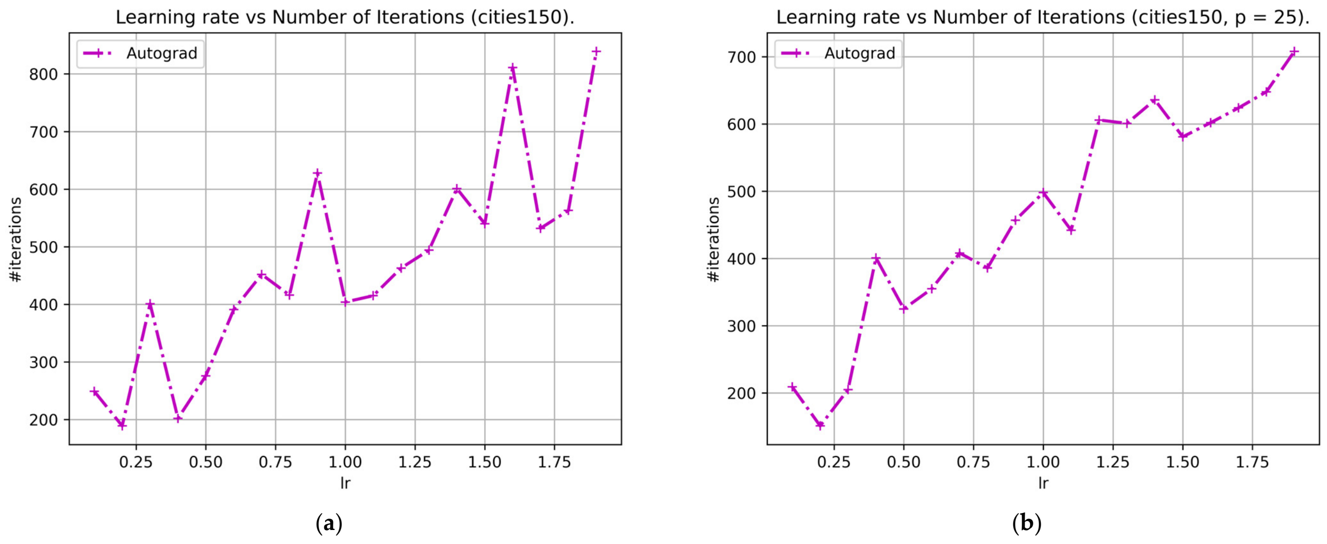


Figure 9. Learning rate vs. the number of iterations (cities150).

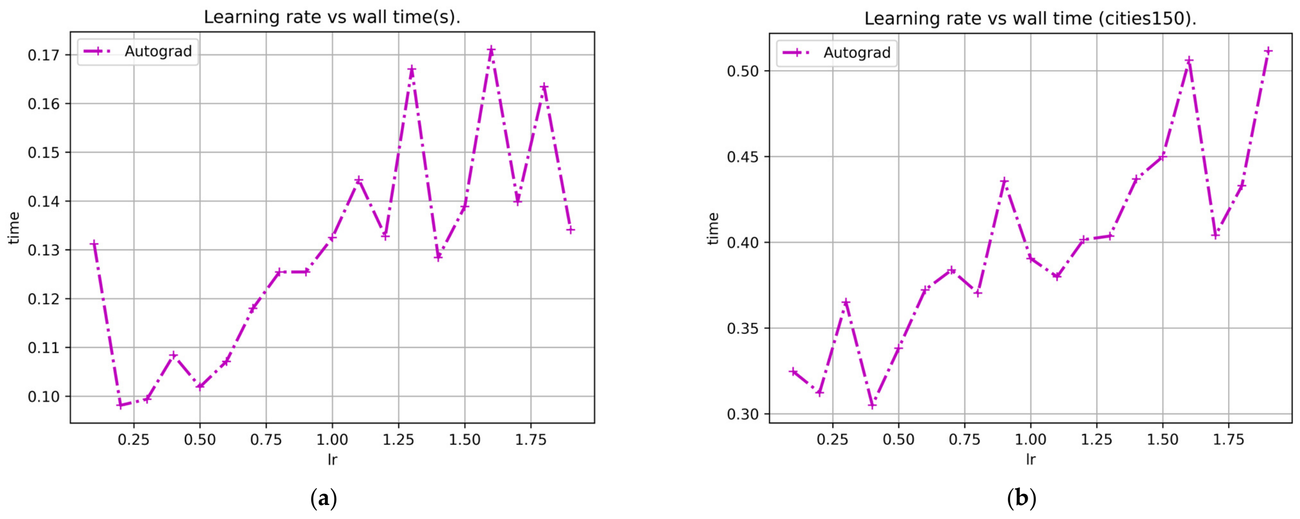


Figure 10. Learning rate vs. computational time (s).

5. Conclusion and Future Work

Lagrangian relaxation involves transforming the original optimization problem into a Lagrangian dual problem by relaxing some or all of its constraints. It is a promising method for solving complex (spatial) optimization problems in that it is both providing proof of optimality (like mathematical programming) and fast (like heuristic procedures). Additionally, it is flexible in that many different relaxations can be created by relaxing different constraint sets. Conventionally, one of the issues with Lagrangian relaxation is the complexity associated with developing an efficient procedure for solving the Lagrangian dual problem and with the derivation and computation of the gradients associated with this procedure. In this article, we demonstrate that this complexity can be alleviated, at least in part, by employing automatic gradient (autograd) libraries originating from the Deep Learning research.

The main idea is that, since autograd libraries can keep track of the computational graph of the Lagrangian function and compute its gradient automatically, the analyst only needs to develop the solution procedure for the Lagrangian function itself in a scientific computing language and test the Lagrangian relaxation algorithm. Since autograd libraries can handle arbitrary loop and conditional statements, the range of Lagrangian problems that can be implemented and fed to autograd computation is quite large. Thus, one may significantly reduce the length of the development cycle of a Lagrangian-relaxation-based algorithm, and allow for more relaxation schemes to be tested.

We demonstrated the feasibility of using autograd for Lagrangian relaxation by implementing it in a subgradient optimization framework for the classic *p*-median problem using both hand-crafted gradients and automatically computed gradients. Our experimental results with the widely used Swain 55 node dataset and a US Cities dataset showed that the autograd version of the Lagrangian algorithm generated exactly the same solutions (final and intermediate) as the primitive version (using hand-crafted gradients). In addition, the optimal solutions from the Lagrangian algorithms are identical to those obtained using an Integer Linear Programming solver. In terms of computational cost, the autograd component did come with a (modest) cost for building the computational graphs.

Overall, we demonstrated the effectiveness of using automatic gradient computation (autograd) in Lagrangian relaxation, using the *p*-median problem as an example. The *p*-median problem is a foundational problem in location-allocation modeling. As shown by Hillsman [12], it can be used as a unifying location model that subsumes other location models as its special cases. In practice, a heuristic solver for the p-median problem is also used as a unifying model in the ESRI Network Analyst's location-allocation module. Consequently, the presented methodology can potentially be applied in GIS systems as an alternative solver with optimality proofs. In addition, we expect that the proposed method can be applied to a wide range of other optimization problems, thereby reducing the cost of development of Lagrangian relaxation algorithms and increasing the domain of its application. This aspect is left for future research.

Author Contributions: Conceptualization, Ting L. Lei; Methodology, Zhen Lei and Ting L. Lei; Software, Zhen Lei and Ting L. Lei; Validation, Zhen Lei and Ting L. Lei; Formal analysis, Zhen Lei and Ting L. Lei; Investigation, Zhen Lei and Ting L. Lei; Resources, Ting L. Lei; Data curation, Zhen Lei and Ting L. Lei; Writing—original draft, Zhen Lei and Ting L. Lei; Writing—review & editing, Zhen Lei and Ting L. Lei; Visualization, Zhen Lei and Ting L. Lei; Supervision, Zhen Lei and Ting L. Lei; Project administration, Ting L. Lei; Funding acquisition, Ting L. Lei. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partly supported by Natural Science Foundation, Grant number BCS-2215155. This research was partly supported by National Natural Science Foundation of China (NSFC) Grant number 41971334.

Data Availability Statement: The data that support the findings of this study are available upon reasonable request.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Hakimi, S.L. Optimum locations of switching centers and the absolute centers and medians of a graph. *Oper. Res.* **1964**, *12*, 450–459. [CrossRef]
2. Hakimi, S.L. Optimum distribution of switching centers in a communication network and some related graph theoretic problems. *Oper. Res.* **1965**, *13*, 462–475. [CrossRef]
3. Fisher, M.L. The lagrangian relaxation method for solving integer programming problems. *Manag. Sci.* **1981**, *27*, 1–18. [CrossRef]
4. Hanjoul, P.; Peeters, D. A comparison of two dual-based procedures for solving the p-median problem. *Eur. J. Oper. Res.* **1985**, *20*, 387–396. [CrossRef]

5. Boyd, S.P.; Vandenberghe, L. *Convex Optimization*; Cambridge University Press: Cambridge, UK, 2004.
6. ReVelle, C.; Marks, D.; Liebman, J.C. An analysis of private and public sector location models. *Manag. Sci.* **1970**, *16*, 692–707. [CrossRef]
7. Greenberg, H.J. The one-dimensional generalized lagrange multiplier problem. *Oper. Res.* **1977**, *25*, 338–345. [CrossRef]
8. Maclaurin, D.; Duvenaud, D.; Johnson, M. Autograd 1.6.2. 2024. Available online: https://pypi.org/project/autograd/ (accessed on 10 December 2024).
9. Lei, T.L. Integrating GIS and location modeling: A relational approach. *Trans. GIS* **2021**, *25*, 1693–1715. [CrossRef]
10. Swain, R.W. A Decomposition Algorithm for a Class of Facility Location Problems. Ph.D. Thesis, Cornell University, New York, NY, USA, 1971.
11. Lei, T.L. Location Modeling Utilizing Closest and Generalized Closest Transport/Interaction Assignment Constructs. Ph.D. Dissertation, University of California, Santa Barbara, CA, USA, 2010.
12. Hillsman, E.L. The p-median structure as a unified linear model for location-allocation analysis. *Environ. Plan. A* **1984**, *16*, 305–318. [CrossRef]