algorithms

MDPI

Article

A Constructive Heuristics and an Iterated Neighborhood Search Procedure to Solve the Cost-Balanced Path Problem

Daniela Ambrosino *,†, **Carmine Cerrone** †, **and Anna Sciomachen** †

Department of Economics and Business Studies, University of Genoa, 16126 Genoa, Italy
* Correspondence: ambrosin@economia.unige.it
† These authors contributed equally to this work.

Abstract: This paper presents a new heuristic algorithm tailored to solve large instances of an NP-hard variant of the shortest path problem, denoted the cost-balanced path problem, recently proposed in the literature. The problem consists in finding the origin–destination path in a direct graph, having both negative and positive weights associated with the arcs, such that the total sum of the weights of the selected arcs is as close to zero as possible. At least to the authors' knowledge, there are no solution algorithms for facing this problem. The proposed algorithm integrates a constructive procedure and an improvement procedure, and it is validated thanks to the implementation of an iterated neighborhood search procedure. The reported numerical experimentation shows that the proposed algorithm is computationally very efficient. In particular, the proposed algorithm is most suitable in the case of large instances where it is possible to prove the existence of a perfectly balanced path and thus the optimality of the solution by finding a good percentage of optimal solutions in negligible computational time.

Keywords: shortest path problem; NP hard combinatorial optimization problem; cost balanced optimization problem; constructive heuristic; neighborhood search procedure

Citation: Ambrosino, D.; Cerrone, C.; Sciomachen, A. A Constructive Heuristics and an Iterated Neighborhood Search Procedure to Solve the Cost-Balanced Path Problem. *Algorithms* **2022**, *15*, 364. https://doi.org/10.3390/a15100364

Academic Editor: Roberto Montemanni

Received: 31 July 2022
Accepted: 27 September 2022
Published: 29 September 2022

1. Introduction

In many optimization problems arising in the decision science area, given a finite set E of elements having a vector cost c associated with them, the objective function is to find a feasible subset $F \subseteq E$, such that the difference in value between the most costly and the least costly selected elements is minimized. Such problems are denoted as balanced optimization problems (BOPs). One of the first approaches to solve a BOP is presented in [1], where the authors consider the set E as an $n \times n$ assignment matrix, the cost c_e as the value contained in a generic cell e of the assignment matrix, and the subset F the selected cells of the defined assignment, thus obtaining the balanced assignment problem. Successively, Ref. [2] introduced the minimum deviation problem, which minimizes the difference between the maximum and average weights in a solution. In the same paper, the authors proposed a general solution scheme also suitable for BOPs. Since then, numerous real-world applications have been defined and solved as BOPs, always with the goal of minimizing the deviation between the highest and lowest cost, or making the performance indices of interest as equal as possible. In the latter case, the most proposed applications in the literature as BOPs include, though are not limited to, production line operations in manufacturing systems (e.g., see [3–6] among others) and supply chain management [7,8]. Other BOPs were proposed in the class of flow and routing problems on networks, where the goal is to design either a single path or a subset of paths in which the arcs or nodes belonging to the solution have a homogeneous value of their relative weights, such as travel time, length, etc. (see [9–11], among others).

This paper deals with a variant of the Shortest Path Problem (SPP) that fits into the class of BOPs. This problem, recently introduced in the literature [12], is denoted Cost Balanced Path Problem ($CBPP$).

The $CBPP$ is defined on a directed weighted graph $G(N, A)$, where N is the set of nodes and A is the set of directed arcs. For each arc $(i, j) \in A$ there is a weight $c_{ij} \in \mathbb{R}$; the weights represent either the increment (in case of positive weight) or the decrement (in case of negative weight) of the cost to balance along the path. The problem is to select in graph G a path p from an origin node o to a destination node d that minimizes the absolute value of the sum of the weights of path p. More formally, the objective function of the $CBPP$ is given by:

$$MIN \ |c(p)| = |\sum_{(i,j)\in p} c_{ij}| \quad (1)$$

In [12] the authors demonstrate that the $CBPP$, as many variants of the SPP [13], is NP-hard in its general form. Moreover, they demonstrate that, in the following two particular cases, the $CBPP$ can be solved in polynomial time:

- When all costs are non-negative or non-positive (i.e., $c_{ij} \geq 0$ or $c_{ij} \leq 0$, $\forall (i, j) \in A$). In this case, the problem is equivalent to the SPP;
- When the cost of the arc is a function of the elevation difference of the two nodes associated with the arc.

The reader may observe that the objective function (1) imposes minimization of an always positive value. For this reason, zero is a lower bound for the CBPP. This implies that any solution with zero objective function value, coincident with the lower bound, will always be optimal. The $CBPP$ can be used for modelling various real problems that require the decision-maker to choose a path in a graph from an origin node to a destination one balancing some elements. Among others, the problems of route design for automated guided vehicles, the vehicles' loading in pick-up and delivery, and storage and retrieval problems in warehouses and yards deserve attention. Another interesting application of the $CBPP$ concerns electric vehicles with limited battery levels, considering, for example, that maintaining a lithium battery at a charge value of about 80% helps to extend its life. In this case, given a graph, the positive or negative values associated with arcs represent, respectively, energy consumption or recharging obtained on downhill roads or through charging stations. Identifying a path in this graph so that the car arrives at its destination with a similar level of charge as at the start would extend the average life of a battery.

In [12], the authors proposed a mixed-integer linear programming model for solving the $CBPP$ and tested it by using different sets of random instances. Experimental tests on the model showed that the computation time of instances that do not have zero as the optimal solution is significantly higher. For these more complex instances, it is, therefore, necessary to develop heuristic algorithms. Unfortunately, at least to the authors' knowledge, no solution approaches for the $CBPP$ have been proposed in the literature. Instead, heuristic algorithms have been presented for some problems closely related to the $CBPP$.

One of the most relevant and similar problem to the $CBPP$ is the cost-balanced Travelling Salesman Problem (TSP) introduced in [14], in which the main objective is to find a Hamiltonian cycle in a graph with total travel cost as close to zero as possible. In [15], the authors propose a variable neighborhood search algorithm, which is a local search with multiple neighborhood structures, to solve the cost-balanced TSP. The balanced TSP is widely described in [16], where an equitable distribution of resources is the main objective. The authors cited many balanced combinatorial optimization problems studied in the literature, and proposed four different heuristics, derived by the double-threshold algorithm and the bottleneck one, to solve the balanced TSP. To solve the same problem, in [17] an adaptive iterated local search is proposed with a perturbation and a random restart that can help in escaping local optimum. In [18], a multiple balanced TSP is analyzed to model and optimize the problems with multiple objectives (salesmen). The goal is to find m Hamiltonian cycles in a graph G by minimizing the difference between the highest edge cost and the smallest edge cost in the tours.

Another problem on graphs closely related to the $CBPP$ is the search for the balanced trees [19], defined as the most appropriate structures (precisely the balanced tree structures) for managing networks with the aim of balancing two or more objectives. In [20], the

authors face the problem of finding two paths in a tree with positive and negative weights. They presented two polynomial algorithms with the goal of minimizing, respectively, the sum of the minimum weighted distances from every node of the tree to the two paths, and the sum of the weighted minimum distances from every node of the tree to the two paths.

To cover the above-mentioned lack of efficient solution methods for the $CBPP$, the main aim of this paper is to propose a heuristic algorithm able to solve large instances of the problem. Since it is not possible to make comparisons with other heuristics proposed in the literature for the $CBPP$, to validate the computational results, an iterative heuristic algorithm is presented. Thus, in the present paper, two heuristic approaches are described, tested, and compared. The first is a two-step heuristic algorithm that implements a constructive heuristic algorithm in the first step (CHa) followed by an improvement phase (IPa). The constructive step is a modified version of the well-known Dijkstra algorithm to solve SPP [21], made necessary to deal with both positive and negative costs and the need to minimize the absolute value of the cost of the path. The second step of the algorithm IPa, starting from the feasible solution provided by CHa, tries to improve it using the information stored by IPa. In particular, starting from the destination node, algorithm IPa goes forward to evaluate new nodes that do not belong to the best-found path in CHa. A similar approach operating forward from the destination node is adopted in the definition of urban multimodal networks [22]. Note that the Dijkstra algorithm has often been combined in solution approaches for solving some variants of the SPP, as, for example, in [23–25]. The second heuristic algorithm is an iterative neighborhood search procedure [26] based on an initial randomly generated solution, improved thanks to IPa previously cited. Among the hundred generated starting solutions, the best one obtained is given for comparison with the previous method.

The computation efficiency of proposed algorithms is tested with randomly generated instances of different sizes, densities, and arc weights. Since no other tests obtained with previously proposed heuristics to solve $CBPP$ are available as validation of the proposed algorithms, we have compared them.

The remaining of this paper is organized as follows. Section 2 presents the proposed algorithms for the $CBPP$, while Section 3 reports the computational experiments. Section 4 gives some conclusions and perspectives.

2. The Heuristic Algorithms

In the following, the two proposed heuristic algorithms are described.

2.1. The Two-Step Algorithm: Constructive and Improvement (CH_IPa)

As already said, the first proposed algorithm to solve the problem under investigation consists of two steps. The first step CHa is a constructive approach based on the Dijkstra algorithm [21]. This algorithm has in input a graph $G(N, A)$, an origin node o and a destination node d, and returns for each node of G the minimum cost for reaching it from node o. The second step IPa is an improvement algorithm that, starting from the feasible solution obtained by CHa, tries to improve it by using as input the solution provided by CHa related to the shortest path tree from the origin node to each node of the graph. Let us describe CH_IPa in more detail.

2.1.1. CHa: Constructive Heuristic

CHa is designed to identify an acyclic path from an origin o node to a destination node d in which the sum of the arcs of the path is as close to zero as possible. This heuristic algorithm follows the scheme of the algorithm for determining the shortest path proposed by Dijkstra [21].

CHa is described in pseudocode in Algorithm 1. Looking at Algorithm 1, the reader can note that one of the differences with respect to Dijkstra's algorithm is due to the extraction criteria of a node, as reported in line 9. In fact, we extract the lowest cost element in absolute value from Q, where Q denotes the set of nodes not yet analyzed.

In particular, the extraction of destination node d from Q (see again line 9 of Algorithm 1), is performed only if d is the only node with a non-infinite cost. This allows the procedure to update the predecessor of node d as many times as possible in order to try to improve the current solution.

Algorithm 1 CHa $(G(N, A), o, d)$.

1: $Q \leftarrow \emptyset$
2: **for** each node $n \in N$ **do**
3: $cost[n] \leftarrow \infty$
4: $prev[n] \leftarrow NULL$
5: $Q \leftarrow Q \cup \{n\}$
6: **end for**
7: $cost[o] \leftarrow 0$
8: **while** $Q \neq \emptyset$ **do**
9: $u \leftarrow$ node in Q with minimum $|cost[u]|$ ▷ (extract d from Q only if $\nexists\, n' \in Q/\{d\} : cost[n'] < \infty$)
10: $Q \leftarrow Q \setminus \{u\}$
11: **for** each neighbor n of u **do**
12: **if** $n \in Q$ **then**
13: $d \leftarrow cost[u] + c_{un}$
14: **if** $|d| < |cost[n]|$ **then**
15: $cost[n] \leftarrow d$
16: $prev[n] \leftarrow u$
17: **end if**
18: **end if**
19: **end for**
20: **end while**
21: **return** $cost$, $prev$

In our algorithm, we examine the set Q of nodes as reported in line 12 of Algorithm 1, only if it is still present in Q to avoid processing the same node several times. This is implicitly done in Dijkstra's algorithm because if a node has already been extracted from Q, it is not possible to reach it later with a lower cost.

Finally, as reported in line 14 of Algorithm 1, in the cost path evaluation from the origin node o, we compare the costs of reaching a node using the absolute value associated with the corresponding arc and select the minimum one.

Just to give an example of CHa, suppose to have to solve the $CBPP$ from node 1 to node 6 of the graph reported in Figure 1. In the same figure are depicted two iterations of the algorithm. The first node selected from Q is node 2, with cost $c_2 = 4$. The next selection is for node 3, with cost $c_3 = |-5| = 5$ (that is less than the cost for reaching node 4, i.e., $c_4 = c_2 + c_{2,4} = |6| = 6$). With two more iterations, reported in Figure 2, node 6 is selected from Q, and a feasible solution is found. The cost for reaching node 6 is 9, and the selected path is nodes 1-2-4-6. Looking at the graph, it is possible to find a better solution, that is, the path consisting of nodes 1-3-4-6 with a cost of 3. We try to improve the obtained solution (nodes 1-2-4-6) by using the improvement algorithm described in the next section.

Using a priority queue proposed in [27], the complexity of Dijkstra's algorithm is equal to $O(|A| + |N|log|N|)$. Considering that the number of nodes extracted from Q does not change in our implementation, using a Fibonacci heap to represent Q, the computational complexity of CHa remains unchanged.

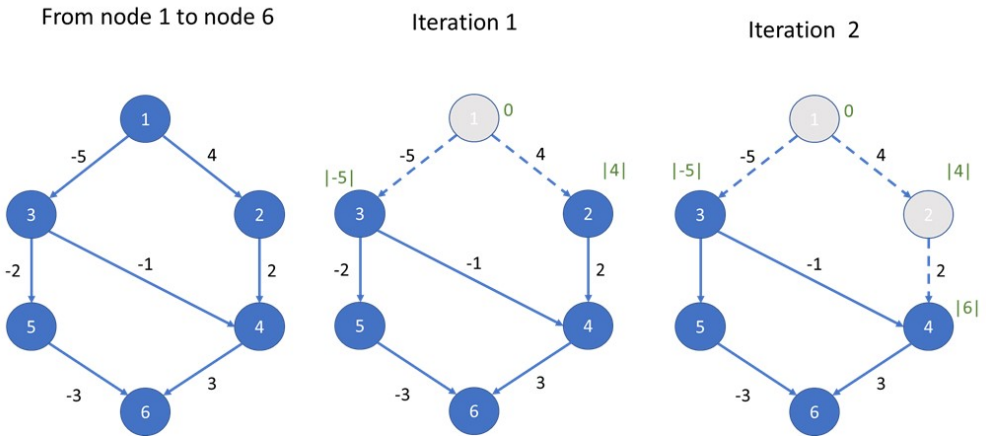

Figure 1. A simple example of two iterations of *CHa*.

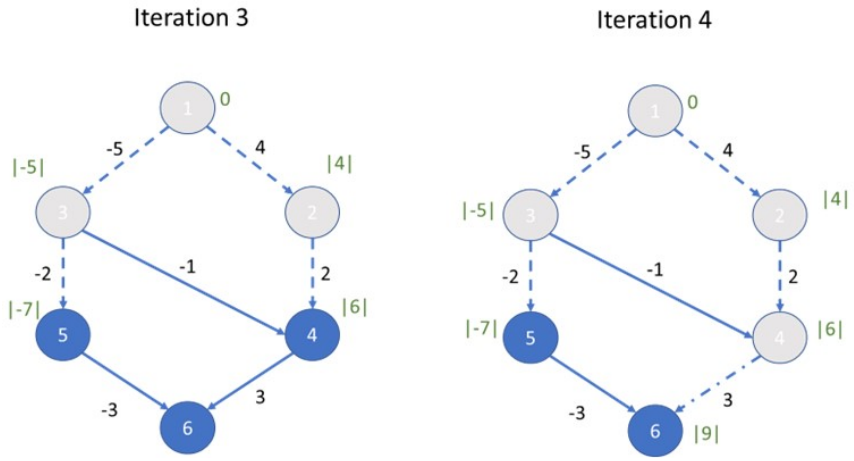

Figure 2. The last iterations of *CHa*.

The last part of the constructive algorithm consists of the identification of a path and its cost evaluation. These procedures are described in Algorithms 2 and 3, respectively. In more detail, the Algorithm 2, here denoted as *Path*, is used to create a set P containing the nodes along the path that connects node o to node d within the tree previously created by *CHa*. The Algorithm 3 *CostPath* is used to compute the sum of the costs of the arcs along the path that connects node o to node d within this tree. Considering that, in the worst case, both Algorithms 2 and 3 must visit all the nodes of the graph $G(N, A)$, their computational complexity is $O(|N|)$.

Algorithm 2 Path (o, d, $prev$).

1: $n \leftarrow d$
2: $P \leftarrow \{n\}$
3: **repeat**
4: $\quad n \leftarrow prev[n]$
5: $\quad P \leftarrow P \cup \{n\}$
6: **until** $n \neq o$
7: **return** P

Algorithm 3 CostPath (o, d, $prev$).

1: $n \leftarrow d$
2: $cost \leftarrow 0$
3: **repeat**
4: $\quad n' \leftarrow prev[n]$
5: $\quad cost \leftarrow cost + c_{n'n}$
6: $\quad n \leftarrow n'$
7: **until** $n \neq o$
8: **return** $cost$

2.1.2. IPa: Improvement Phase

The improvement Algorithm 4, here denoted IPa, is described below. The algorithm has been designed to improve the solution obtained by the constructive heuristic CHa, by exploiting the tree stored in $prev$. In particular, this algorithm inversely visits the path from o to d produced by CHa. Starting at node d (line 1), it enters a loop (line 3) that ends only when node o is reached. For each node n in the path, it tries to identify a new parent n' (line 6) which allows it to improve the objective function (line 7). To avoid the creation of sub-cycles, it is tested that the new parent n' is not a descendant of the current node n (line 8).

Algorithm 4 IPa ($G(N, A)$, o, d, $cost$, $prev$).

1: $n \leftarrow d$
2: $currCost \leftarrow cost[d]$
3: **while** $n \neq o$ **do**
4: $\quad np \leftarrow prev[n]$
5: $\quad$ **for** each arc $(n', n) \in A$ **do**
6: $\quad\quad ds \leftarrow cost[n'] + c_{n'n} + CostPath(n, d, prev)$
7: $\quad\quad$ **if** $|ds| < |currCost|$ **then**
8: $\quad\quad\quad$ **if** $Path(n, d, prev) \cap Path(o, n', prev) = \emptyset$ **then**
9: $\quad\quad\quad\quad np \leftarrow n'$
10: $\quad\quad\quad$ **end if**
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad prev[n] \leftarrow np$
14: $\quad n \leftarrow np$
15: **end while**
16: **return** $prev, currCost$

Analyzing the computational complexity of this algorithm, we have a complexity of $O(|N|)$ in line 3, having to iterate over a maximum of $|N|$ nodes, and for the loop of line 5 forces the algorithm to visit a maximum of $(|N| - 1)$ arcs, reaching a complexity of $O(|N|^2)$. Finally, considering the complexity of the functions $Path$ and $CostPath$ we reach a computational complexity equal to $O(|N|^3)$.

Just to give an idea of the IPa, in Figure 3 the starting solution and the first improvement iteration are reported.

In particular, starting from node 6, the algorithm searches for a node connected to 6, reached during the execution of CHa and not belonging to the current solution. The first (and unique) candidate is node 5. Before accepting node 5 and modifying the solution, the algorithm checks if passing through node 5 to reach node 6 from node 1 is cheaper than the current solution. This is not the case, thus node 5 is not selected. Moreover, going forward from node 6 to node 4, there is a candidate node 3. Connecting node 3 to 4 is convenient: the new path from the origin node 1 to node 3, plus the cost of the new arc connecting node 3 to node 4, and the cost from node 4 to the destination node 6 permits to improve the objective function that passes from |9| to |3|. This selection is accepted and the search continues going forward from node 3 to the origin. There are no more possibilities to improve the solution, thus the algorithm stops.

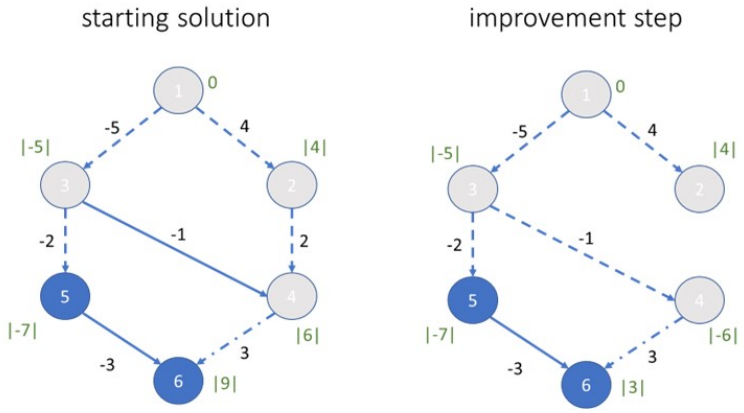

Figure 3. An example of *IPa*.

2.2. The Iterated Neighborhood Search Procedure

The second heuristic algorithm to solve the *CBPP* is an iterative neighborhood search procedure. A feasible solution is randomly generated and then it is improved. In the improvement phase, we use the improvement algorithm defined in Section 2.1.2. The obtained solution is stored and the process is iterated many times starting from a new random generated feasible solution. The iterated neighborhood search Algorithm 5, called *RP*, describes in detail the generation of the starting feasible solution. It identifies a random path between the origin node o and the destination node d, and creates a random spanning tree rooted in the origin node. The whole iterated process is described in Algorithm 6, called *RPR*. Note that $it = 100$ different starting solutions are generated. At the end, the best improved solution is returned and used for the comparison with the previous method. Algorithm 7 describes the *RandomTree* function used within the *RP* and *RPR* function code to generate a randomized rooted tree.

Algorithm 5 RP $(G(N, A), o, d, it)$.

$P \leftarrow \emptyset$
$cost \leftarrow \infty$
for it iterations **do**
  $cost, prev \leftarrow RandomTree(G, o, d)$
  if $|CostPath(o, d, prev)| < |cost|$ **then**
    $cost \leftarrow CostPath(o, d, prev)$
    $P \leftarrow Path(o, d, prev)$
  end if
end for
return P

Algorithm 6 RPR $(G(N, A), o, d, it)$.

$P \leftarrow \emptyset$
$cost \leftarrow \infty$
for it iterations **do**
  $cost, prev \leftarrow RandomTree(G, o, d)$
  $currCost, prev \leftarrow IP(G, o, d, cost, prev)$
  if $|CostPath(o, d, prev)| < |cost|$ **then**
    $cost \leftarrow CostPath(o, d, prev)$
    $P \leftarrow Path(o, d, prev)$
  end if
end for
return P

Algorithm 7 RandomTree ($G(N, A)$, o, d).

$Q \leftarrow \emptyset$
for each node $n \in N$ **do**
    $cost[n] \leftarrow \infty$
    $prev[n] \leftarrow NULL$
    $Q \leftarrow Q \cup \{n\}$
end for
$cost[o] \leftarrow 0$
while $Q \neq \emptyset$ **do**
    $u \leftarrow$ random node in Q
    $Q \leftarrow Q \setminus \{u\}$
    for each neighbor n of u **do**
        if $cost[n] = \infty$ **then**
            $d \leftarrow cost[u] + c_{un}$
            $cost[n] \leftarrow d$
            $prev[n] \leftarrow u$
        end if
    end for
end while
return $cost$, $prev$

3. Computational Experiments

In this section, we report the computational experimentation performed to validate the proposed algorithms. The computational tests were performed on a MacBook Pro (Apple Inc., Cupertino, CA, USA), with a 2.9 GHz Intel i9 (Intel Inc., Santa Clara, CA, USA) processor and 32 GB of RAM. In all tests, we used as input the two sets of instances reported in [12] and some new large random instances also generated, as described in [12].

The first set of instances, named *Grid*, is characterized by complete square grids, where each node is connected to its four neighbors. The second set of instances, named *Rand*, is characterized by randomly generated connected graphs with the average degree ranging from 2 to 20. All instances used in this section can be found at the link [28]. In the following, we will refer to these instances as $Grid[Rand] - n1 - n2$, where $n1$ represents the number of nodes and $n2$ represents the percentage of arcs incident on each vertex. The third set of instances is an extension of the *Rand* instances generated with the same criteria but with a number of nodes equal to 500, 1000, 2000, and 5000.

The costs associated with the arcs of each instance were generated following 3 different schemes, based on a homogeneous distribution of randomly generated costs with different ranges, respectively, $[-10, 10]$, $[-100, 100]$ and $[-1K, 1K]$.

Each row in the following tables reports the average of the results of five instances.

In the following different experimental campaigns are reported.

3.1. First Experimental Campaign: Small Instances

Small instances have been optimally solved by the mathematical model presented in [12], thus we are able to compare the results obtained by the proposed CH_IPa algorithm with the optimal solutions. Thanks to the following results, it is possible to understand the behaviour of the proposed algorithm, in particular the effectiveness of IPa, and to compare *Grid* and *Rand* instances.

Table 1 shows the number of optimal solutions obtained, respectively, using CHa and CH_IPa. Looking at Table 1, we can observe that IPa always improves the solutions obtained by CHa. This improvement is more evident for instances with costs $[-10, 10]$, while the number of optimal solutions remains almost the same for $[-1K, 1K]$ instances. The two sets of instances (*Grid* and *Rand*) have the same behavior.

All instances can be solved in less than one millisecond.

Table 1. Number of optimal solutions identified—instances with random costs.

Instance	*CHa*			*CH_IPa*		
	[−10, 10]	**[−100, 100]**	**[−1K, 1K]**	**[−10, 10]**	**[−100, 100]**	**[−1K, 1K]**
Grid_100_10	2	0	0	2	1	0
Grid_225_15	0	0	0	3	1	0
Grid_400_20	1	0	0	5	0	0
# OPT	**3**	**0**	**0**	**10**	**2**	**0**
Rand_100_02	0	0	0	1	0	0
Rand_100_03	0	1	1	3	1	1
Rand_100_04	1	1	0	4	1	0
Rand_100_05	2	0	0	3	1	0
Rand_100_10	3	1	0	5	3	0
Rand_100_20	5	0	0	5	2	0
# OPT	**11**	**3**	**1**	**21**	**8**	**1**
Rand_200_02	2	0	0	3	1	0
Rand_200_03	2	0	0	4	1	0
Rand_200_04	0	0	0	5	0	0
Rand_200_05	2	0	0	4	0	0
Rand_200_10	4	2	0	5	3	0
Rand_200_20	5	1	0	5	2	1
# OPT	**15**	**3**	**0**	**26**	**7**	**1**

Actually, in Table 2 are reported the absolute objective function values obtained at the end of *CHa* and *CH_IPa* for instances with randomly generated costs in $[-10, 10]$, $[-100, 100]$ and $[-1K, 1K]$. Note that, the optimal value for all instances is zero (as shown in [12]). From Table 2, it is evident that it is always possible to improve the objective function values using *IPa* after *CHa*. In particular, for the largest instances (i.e., last row of each group), both Grid and Rand, with costs in $[-10, 10]$, *IPa* is able to provide optimal solutions. On average, for grid instances, the improvements are 92.3%, 92.2% and 87.8%, respectively, for instances with costs $[-10, 10]$, $[-100, 100]$, and $[-1K, 1K]$. The improvements are about 70% for instances of type Rand_100, and about 82% for Rand_100. In all cases, the improvement is lower when costs are in $[-1K, 1K]$.

Table 2. Absolute objective function values obtained—instances with random costs.

Instance	*CHa*			*CH_IPa*		
	[−10, 10]	**[−100, 100]**	**[−1K, 1K]**	**[−10, 10]**	**[−100, 100]**	**[−1K, 1K]**
Grid_100_10	3.6	20.8	393.8	0.6	3.4	58.0
Grid_225_15	4.2	57.2	506.0	0.4	4.4	53.2
Grid_400_20	4.0	57.0	448.6	0.0	2.8	53.2
AVG	**3.9**	**45.0**	**449.5**	**0.3**	**3.5**	**54.8**
Rand_100_02	5.0	54.0	782.2	1.4	15.2	249.0
Rand_100_03	4.0	40.8	253.6	2.0	5.6	62.6
Rand_100_04	3.4	21.2	225.2	0.2	4.4	49.0
Rand_100_05	0.8	12.0	239.6	0.6	4.0	64.4
Rand_100_10	0.4	4.2	40.4	0.0	0.8	37.4
Rand_100_20	0.0	5.2	41.2	0.0	1.2	12.8
AVG	**2.3**	**22.9**	**263.7**	**0.7**	**5.2**	**79.2**
Rand_200_02	2.0	30.2	314.4	0.6	6.6	41.6
Rand_200_03	0.8	18.2	237.0	0.2	9.4	123.2
Rand_200_04	2.2	17.0	128.2	0.0	2.2	10.4
Rand_200_05	1.4	15.0	73.2	0.2	1.0	4.6
Rand_200_10	0.2	3.0	31.0	0.0	1.2	12.6
Rand_200_20	0.0	0.8	24.8	0.0	0.6	3.8
AVG	**1.1**	**14.0**	**134.8**	**0.2**	**3.5**	**32.7**

Finally, by comparing the results shown in Table 2 with the optimal values, that is for all these instances equal to zero, we can note that the optimality gap is larger for instances with costs in $[-1K, 1K]$.

In further computational experimentation, we solved instances whose optimal value of the objective function is not zero. In particular, the costs associated with the arcs of the corresponding graph are obtained as follows: a random cost is associated with each node in the range [0,10,000]. Then, the cost of an arc is obtained by a 1% random perturbation of the displacement of the costs. We will refer to this cost structure as *P–EL*.

For this type of instance, we noted that the proposed algorithm did not provide good solutions, even if it performs very quickly. The best case corresponds to the Rand _100 _03 set of instances, for which the optimal value is 6005, while the solutions obtained at the end of CHa and CH_IPa, respectively, are 6557 and 5431, thus corresponding to an optimality gap of 7%. Unfortunately, in the worst case, we have a set of instances (Rand_200_20) with the average optimal function value equal to 74, while our algorithm is not able to go down 2605. Note that, also in these cases, the computational time is negligible.

The above computational results suggest that the proposed algorithm can produce effective solutions in case of very large instances and when the cost of the arcs is generated uniformly. Instead, it appears that the algorithm cannot be used for instances with cost-structure *P–EL*. It also seems that as the size of the instances increases, the quality of the solution produced improves, as does the number of optimal solutions identified.

3.2. Second Experimental Campaign: Large Instances

This experimental campaign is based on the third set of generated instances, a set of random, larger-sized instances, for which it is not possible to obtain the optimal solution by the mathematical model used to solve small instances.

These tests permit a better investigation of the behavior of the proposed CH_IPa algorithm. The obtained results are compared with those of the iterated neighborhood search procedure. In these new sets of instances, the number of nodes ranges from 500 to 5000. For completeness, the following tables report also the results related to small instances. Note that tests have been executed with costs generated uniformly ($[-10, 10]$, $[-100, 100]$ and $[1K, 1K]$) and with cost structure *P–EL*.

Table 3 shows the behavior of CH_IPa as the size of the instances increases. The number of solutions with objective function values equal to zero is shown. Although we do not know the optimal solution for the generated large instances, obviously we can certify the optimality of the heuristic solution in case a solution with an objective function value equal to zero is identified. We can see that, as the size of the instance increases, the number of zero-solutions improves. The table shows that the proposed algorithm is able to identify 92% of optimal solutions for the solved instances (i.e., 167 on 180) in case of uniform cost distribution in $[-10, 10]$. In all instances with a number of nodes greater than or equal to 500, we can obtain the optimal solution. Moreover, with the increase in $|N|$ also for the distribution of costs *P–EL* we are able to identify numerous optimal solutions, with $|N| = 5000$ we identify at least 25 optimal solutions (i.e., 83%).

Table 3. Number of optimal solutions identified by CHa and CH_IPa—large instances.

$\|N\|$	CHa				CH_IPa			
	$[-10, 10]$	$[-100, 100]$	$[-1K, 1K]$	*P–EL*	$[-10, 10]$	$[-100, 100]$	$[-1K, 1K]$	*P–EL*
100	11	3	1	0	21	8	1	0
200	15	3	0	0	26	7	1	1
500	24	5	0	0	30	18	3	3
1000	25	14	2	0	30	28	13	6
2000	30	17	2	0	30	30	20	13
5000	30	25	5	0	30	30	27	25
#OPT	**137**	**67**	**10**	**0**	**167**	**121**	**65**	**48**

Table 4 shows the average value of the objective function for the same scenario as Table 3. The data contained in this table are used in Figure 4 to highlight how the solution produced improves as the size increases.

Table 4. Absolute objective function values—CHa and CH_IPa—large instances.

$\|N\|$	CHa				CH_IPa			
	[−10, 10]	**[−100, 100]**	**[−1K, 1K]**	***P–EL***	**[−10, 10]**	**[−100, 100]**	**[−1K, 1K]**	***P–EL***
100	2.3	22.9	263.7	3165.4	0.7	5.2	79.2	3057.5
200	1.1	14.0	134.8	3800.4	0.2	3.5	32.7	3583.8
500	0.6	5.8	45.6	3086.2	0.0	0.4	6.1	2462.2
1000	0.2	2.1	18.1	3782.4	0.0	0.1	1.1	2776.3
2000	0.0	0.8	14.0	3129.2	0.0	0.0	0.5	1374.0
5000	0.0	0.3	3.2	3461.1	0.0	0.0	0.1	272.3
AVG	**0.7**	**7.7**	**79.9**	**3404.1**	**0.1**	**1.5**	**20.0**	**2254.4**

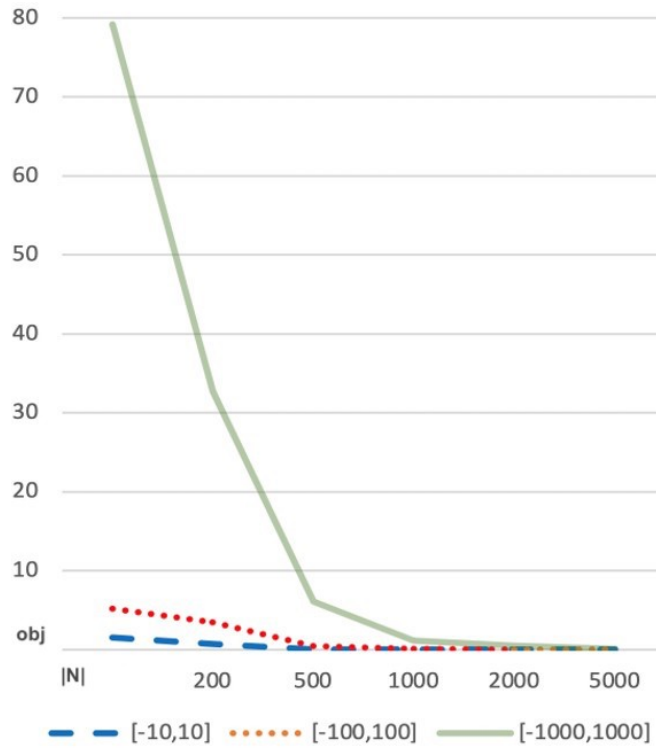

Figure 4. Objective function values obtained by CH_IPa compared to the size of the instances.

Table 5 reports the CPU time in milliseconds. The computational time required to produce a solution using the CH_IPa is less than half a second even for instances with 5000 nodes. These computational times suggest that the average number of iterations of the algorithm is significantly less than the number of iterations associated with the computational complexity of the worst case $O(|N|^3)$. Table 6 shows the average number of iterations performed by IPa. The IOM column is useful to understand how many iterations, on average, are performed for every million theoretical iterations associated with the worst case $O(|N|^3)$. Figure 5 shows the relationship between the number of iterations and the number of nodes in the graph. For the set of instances used, the trend as $|N|$ increases appears linear.

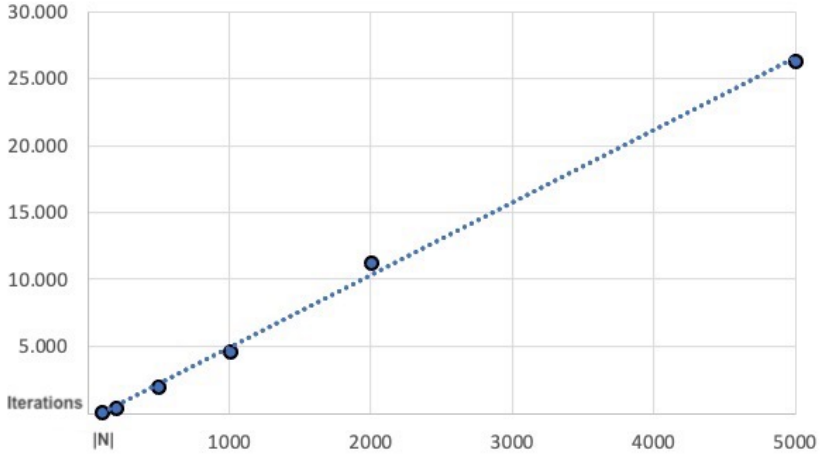

Figure 5. Relationship between the number of IPa iterations and $|N|$.

Table 5. CPU time (milliseconds) of the proposed CH_IPa

Instance	CH_IPa			
	[−10, 10]	**[−100, 100]**	**[−1K, 1K]**	***P–EL***
Rand_100_02	0	0	0	0
Rand_100_03	0	0	0	0
Rand_100_04	0	0	0	0
Rand_100_05	0	0	0	0
Rand_100_10	0	0	0	0
Rand_100_20	0	0	0	0
Rand_200_02	0	0	0	0
Rand_200_03	0	0	0	0
Rand_200_04	0	0	1	0
Rand_200_05	0	0	0	0
Rand_200_10	0	0	1	1
Rand_200_20	0	0	0	1
Rand_500_02	1	0	1	0
Rand_500_03	1	0	0	1
Rand_500_04	0	1	1	1
Rand_500_05	0	1	1	1
Rand_500_10	1	2	1	1
Rand_500_20	3	3	3	3
Rand_1000_02	2	2	2	2
Rand_1000_03	2	3	3	2
Rand_1000_04	2	3	3	3
Rand_1000_05	4	4	4	4
Rand_1000_10	7	8	7	8
Rand_1000_20	11	11	11	12
Rand_2000_02	8	9	9	9
Rand_2000_03	10	12	12	11
Rand_2000_04	12	15	14	14
Rand_2000_05	13	15	15	14
Rand_2000_10	25	27	27	25
Rand_2000_20	48	51	53	55
Rand_5000_02	42	50	52	53
Rand_5000_03	57	67	71	65
Rand_5000_04	73	85	90	83
Rand_5000_05	88	103	112	101
Rand_5000_10	217	204	202	201
Rand_5000_20	366	350	371	328
AVG	**27.6**	**28.5**	**29.6**	**27.7**

Table 6. Average and theoretical *IPa* (worst case) iterations, and CPU time.

$\|N\|$	Iterations	IOM	Time (ms)
100	128	128	0
200	392	49	0
500	1966	16	1
1000	4640	5	7
2000	11,303	1	27
5000	26,305	0	143

In the following tables, the results obtained by CH_IPa are compared with those obtained by using the iterated neighborhood search procedure described in Section 2.2. In each Table, both the best among the first random paths (RP) and the best path after the improvement phase (RPR) are reported.

Tables 7 and 8 show, respectively, the number of optimal solutions and the computational times in milliseconds, identified by CH_IPa, RP and RPR.

It can be seen from these tables that CH_IPa produces the maximum number of optimal solutions in about $\frac{1}{50}$ of the computational time required by the RP and RPR iterative techniques.

In particular, CH_IPa is able to find more 30% optimal solutions than RPR for costs $[-100, 100]$ and P–EL, and about 180% for cost $[-1K, 1K]$.

These results also show that IPa applied to the RP significantly increases the number of optimal solutions identified by the RP, passing from 181 to 319 optimal solutions identified; the greatest effect is on instances with cost P–EL and $[-1K, 1K]$.

Table 7. Comparison of the number of optimal solutions identified.

	$\|N\|$	$[-10, 10]$	$[-100, 100]$	$[-1K, 1K]$	P–EL
CH_IPa	100	21	8	1	0
	200	26	7	1	1
	500	30	18	3	3
	1000	30	28	13	6
	2000	30	30	20	13
	5000	30	30	27	25
	#OPT	**167**	**121**	**65**	**48**
RP	100	25	8	1	0
	200	23	5	2	0
	500	22	5	0	0
	1000	21	7	0	0
	2000	22	7	0	0
	5000	26	5	2	0
	#OPT	**139**	**37**	**5**	**0**
RPR	100	25	11	1	0
	200	27	11	3	1
	500	27	12	5	3
	1000	29	16	2	5
	2000	28	22	3	10
	5000	30	21	9	18
	#OPT	**166**	**93**	**23**	**37**

Table 8. Comparison of the CPU time (milliseconds).

	$\|N\|$	$[-10, 10]$	$[-100, 100]$	$[-1K, 1K]$	*P–EL*
CH_IPa	100	0	0	0	0
	200	0	0	0	0
	500	1	1	1	1
	1000	5	5	5	5
	2000	19	22	22	21
	5000	140	143	150	138
	AVG	**28**	**28**	**30**	**28**
RP	100	1	1	1	1
	200	2	3	3	3
	500	13	24	26	25
	1000	113	164	195	206
	2000	776	1183	1294	1288
	5000	2263	7192	7359	7359
	AVG	**528**	**1428**	**1480**	**1480**
RPR	100	1	1	1	1
	200	3	3	3	3
	500	25	26	26	25
	1000	210	192	195	208
	2000	1315	1224	1300	1289
	5000	8025	7369	7373	7361
	AVG	**1596**	**1469**	**1483**	**1480**

4. Conclusions and Outline Future Works

In this paper, we addressed the Cost-Balanced Path Problem, which fits into the class of the balanced combinatorial optimization problems. A two-step heuristic algorithm is proposed for solving this variant of the classical Shortest Path Problem. An iterative heuristic has been developed to compare the produced solutions. To the authors' knowledge, this is the first research work related to heuristic algorithms to solve the *CBPP*. This two-step heuristic algorithm (*CH_IPa*) is able to find feasible solutions in a negligible computational time and is particularly suitable for large-sized graphs. In fact, it is worth noting that by executing the constructive and the improvement heuristic algorithms consecutively, we can always find a larger number of optimal solutions in a very short CPU time. In particular, comparing the *CH_IPa* solutions with those obtained by the iterative algorithm, which is a simple and fast heuristic method. Since the *CBPP* can be applied in many real-life problems for which only large size instances are required, such as vehicle battery level, altitude change, and cargo problems among others, as future work the authors will work on the development of a metaheuristic to be able to improve the goodness of the solutions for many types of instances of the *CBPP*.

Author Contributions: D.A., C.C. and A.S. contributed equally to this work. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data available on request.

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

Abbreviations

The following abbreviations are used in this manuscript:

SPP	Shortest Path Problem
CBPP	Cost-Balanced Path Problem
TSP	Travelling Salesman Problem
MILP	Mixed Integer Linear Programming
CHa	Constructive Heuristic algorithm
IPa	Improvement Phase algorithm

References

1. Martello, S.; Pulleyblank, W.R.; Toth, P.; de Werra, D. Balanced optimization problems. *Oper. Res. Lett.* **1984**, *3*, 275–278. [CrossRef]
2. Duin, C.W.; Volgenant, A. Minimum deviation and balanced optimization: A unified approach. *Oper. Res. Lett.* **1991**, *10*, 43–48. [CrossRef]
3. McGovern, S.M.; Gupta, S.M. A balancing method and genetic algorithm for disassembly line balancing. *Eur. J. Oper. Res.* **2007**, *179*, 692–708. [CrossRef]
4. Levitin, G.; Rubinovitz, J.; Shnits, B. A genetic algorithm for robotic assembly line balancing. *Eur. J. Oper. Res.* **2006**, *168*, 811–825. [CrossRef]
5. Papadopoulos, H.T.; Vidalis, M.I. Minimizing WIP inventory in reliable production lines. *Int. J. Prod. Econ.* **2001**, *70*, 185–197. [CrossRef]
6. Arbib, C.; Lucertini, M.; Nicolò, F. Workload balance and part-transfer minimization in flexible manufacturing systems. *Int. J. Flex. Manuf. Syst.* **1991**, *3*, 5–25. [CrossRef]
7. Bhagwat, R.; Sharma, M.K. Performance measurement of supply chain management using the analytical hierarchy process. *Prod. Plan. Control.* **2007**, *18*, 666–680. [CrossRef]
8. Babaioff, M.; Walsh, W.E. Incentive-compatible, budget-balanced, yet highly efficient auctions for supply chain formation. *Decis. Support Syst.* **2005**, *39*, 123–149. [CrossRef]
9. Kaspi, M.; Kesselman, U.; Tanchoco, J.M.A. Optimal solution for the flow path design problem of a balanced unidirectional AGV system. *Int. J. Prod. Res.* **2002**, *40*, 389–401. [CrossRef]
10. Chen, H.; Ye, H.Q. Asymptotic optimality of balanced routing. *Oper. Res.* **2012**, *60*, 163–179. [CrossRef]
11. Li, X.; Wei, K.; Aneja, Y.P.; Tian, P.; Cui, Y. Matheuristics for the single-path design-balanced service network design problem. *Comput. Oper. Res.* **2017**, *77*, 141–153. [CrossRef]
12. Ambrosino, D.; Cerrone, C. The Cost-Balanced Path Problem: A Mathematical Formulation and Complexity Analysis. *Mathematics* **2022**, *10*, 804. [CrossRef]
13. Turner, L. Variants of the shortest path problem. *Alg. Oper. Res.* **2011**, *6*, 91–104.
14. Greco, S.; Pavone, M.F.; Talbi, E.-G.; Vigo, D. Metaheuristics for combinatorila optimization. In *Advances in Intelligent Systems and Computing*; Springer: Cham, Switzerland, 2021.
15. Akbay, M.; Kalayci, C. A variable neighborhood search algorithm for cost-balanced travelling salesman problem. In *Metaheuristics for Combinatorila Optimization Advances—Intelligent Systems and Computing*; Springer: Cham, Switzerland, 2021; pp. 23–36.
16. Larusic, J.; Punnen, A.P. The balanced traveling salesman problem. *Comput. Oper. Res.* **2011**, *38*, 868–875. [CrossRef]
17. Pierotti, J.; Ferretti, L; Pozzi, L.; van Essen, J.T. Adaptive Iterated Local Search with Random Restarts for the Balanced Travelling Salesman Problem. In *Metaheuristics for Combinatorila Optimization Advances—Intelligent Systems and Computing*; Springer: Cham, Switzerland, 2021; pp. 36–56.
18. Dong, X.; Xu, M.; Lin, Q.; Han, S.; Li, Q.; Guo, Q. IT algorithm with local search for large scale multiple balanced traveling salesmen problem. *Knowl.-Based Syst.* **2021**, *229*, 107330. [CrossRef]
19. Moharam, R.; Morsy, E. Genetic algorithms to balanced tree structures in graphs. *Swarm Evol. Comput.* **2017**, *32*, 132–139. [CrossRef]
20. Zhou, J.; Kang, L.; Shan, E. Two paths location of a tree with positive or negative weights. *Theor. Comput. Sci.* **2015**, *607*, 296–305. [CrossRef]
21. Dijkstra, E.W. A note on two problems in connexion with graphs. *Numer. Math.* **1959**, *1*, 269–271. [CrossRef]
22. Ambrosino, D.; Sciomachen, A. An Algorithmic Framework for Computing Shortest Routes in Urban Multimodal Networks with Different Criteria, *Procedia Soc. Behav. Sci.* **2014**, *108*, 139–152.
23. Dinitz, Y.; Itzhak, R. Hybrid Bellman–Ford–Dijkstra algorithm. *J. Discret. Alg.* **2017**, *42*, 35–44. [CrossRef]
24. Lewis, R. Algorithms for Finding Shortest Paths in Networks with Vertex Transfer Penalties. *Algorithms* **2020**, *13*, 269. [CrossRef]
25. Abdelghany, H.M.; Zaki, F.W.; Ashour, M.M. Modified Dijkstra Shortest Path Algorithm for SD Networks. *Int. J. Electr. Comput. Eng. Syst.* **2022**, *13*, 203. [CrossRef]
26. Carrabs, F.; Cerrone, C.; Cerulli, R.; Silvestri, S. The rainbow spanning forest problem *Soft Comput.* **2018**, *22*, 2765–2776. [CrossRef]

27. Fredman, M.L.; Tarjan, R.E. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* **1987**, *34*, 596–615. [CrossRef]
28. Test Instances. Available online: https://github.com/CarmineCerrone/CostBalancedPathProblem (accessed on 12 September 2022).