Submitted to *INFORMS Journal on Computing*
manuscript JOC-2015-11-OA-210.R1

A Branch Decomposition Algorithm for the p-Median Problem

Caleb C. Fast, Illya V. Hicks
Department of Computational and Applied Mathematics, Rice University, Houston, Texas 77005 calebfast@rice.edu, ivhicks@rice.edu

In this paper, we use a branch decomposition technique to improve approximations to the p-median problem. Starting from a support graph produced either by a combination of heuristics or by linear programming, we use dynamic programming guided by a branch decomposition of that support graph to find the best p-median solution on the support graph. Our results show that when heuristics are used to build the support graph and the support graph has branchwidth at most 7, our algorithm is able to provide a solution of lower cost than any of the heuristic solutions. When linear programming is used to build the support graph and the support graph has branchwidth at most 7, then our algorithm provides better solutions than popular heuristics and is faster than integer programming. Thus, our algorithm is a useful practical tool when support graphs have branchwidth at most 7.

Key words: Facility Location, p-Median, Branch Decompostion, Dynamic Programming

1. Introduction

Solution approaches to NP-hard problems fall into four basic categories: meta-heuristics, combinatorial approximation algorithms, exact combinatorial algorithms, or integer programming. Since the problems are NP-hard, exact solution methods are often impractical. On the other hand, sub-optimal solutions such as those provided by meta-heuristics and approximation algorithms require practicioners to accept costs that could be avoided with a better solution. In this paper, we use a branch decomposition technique to improve approximations to the p-median problem. This technique can be used to develop a solution from the linear relaxation of the problem, or to combine a pool of heuristic solutions into a solution of higher quality than any in the original pool. Our computational results show that our technique provides solutions of better quality than popular heuristics, while being significantly faster than integer programming on graphs with low branchwidth.

The *p-median problem* (PMP) is exactly like the uncapacitated facility location problem except that the number of facilities is fixed to some positive integer p. Given a set of locations that serve as both customer locations and potential facility locations and given a set of costs associated with serving each customer from each potential facility location, the PMP asks for the set of p facility locations that minimizes the cost of serving the locations that are not chosen. It is useful in a variety of applications where the number of facilities cannot be easily increased due to temporal or political constraints. For example, Kunkel et al. (2014) used the PMP to distribute medical supplies in Malawi. The problem has been attacked with many different solution methods. Heuristic approaches include tabu search by Rolland et al. (1997), random search by Antamoshkin and Kazakovtsev (2013), local search by Arya et al. (2004), and dynamic programming by Hribar and Daskin (1997). Resende and Werneck (2004) also developed a hybrid heuristic with elements of random search, tabu search, local search, greedy search, and other methods. Approximation algorithms with a guaranteed error bound have also been developed. To our knowledge, the current best approximation guarantee is the $1+\sqrt{3}+\epsilon$ bound of Li and Svensson (2013).

In this paper, we develop a heuristic for the PMP using branch decompositions of support graphs produced either by heuristics or by linear programming. Section 2 gives the integer programming formulation of the PMP and general background on branch decompositions. Section 3 introduces our algorithm. We show complexity results and theoretical error bounds in §4. We give computational results comparing our algorithm to other state-of-the-art algorithms in §5, and we offer conclusions in §6.

2. Preliminaries

We will consider the PMP defined on a complete directed graph (that is, a graph where directed edges exist in both directions between each pair of nodes), call it $\vec{K}(V,A)$, with edge costs c. The problem asks for a subset $M \subseteq V$ such that $|M| = p$ and the sum of the distances from each node of $V \backslash M$ to it's closest neighbor in M is minimized. Definition 1 gives the standard, straightforward formulation for the PMP. Alternative formulations exist for the problem. Avella and Sassano (2001) gave a formulation that relates the PMP to the stable set problem. Elloumi (2010) gave a formulation in terms of neighborhoods, which requires less linear constraints if the distance matrix is sparse. However, since we deal with complete graphs in this paper, the distance matrix is dense, and we use the standard formulation.

Definition 1. Integer Program Model of the PMP (PMPIP)

$$\begin{array}{llll} \text{minimize} & c^T x & & \\ \text{subject to:} & \sum_{v \in V} x_{v,v} = p & & (1) \\ & \sum_{v_i \in V} x_{v,v_i} = 1 & \forall v \in V & (2) \\ & x_{v_1,v_2} \leq x_{v_2,v_2} & \forall v_1, v_2 \in V & (3) \\ & 0 \leq x_{v_1,v_2} \leq 1 & \forall v_1, v_2 \in V,\, v_1 \neq v_2 & (4) \\ & x_{v_1,v_2} \in \{0,1\} & \forall v_1, v_2 \in V,\, v_1 = v_2 & (5) \end{array}$$

The linear relaxation of an integer program is the linear program that arises when the integrality constraints of an IP (the constraints (5) in the PMPIP) are relaxed to allow fractional values. We wish to solve the linear relaxation of the PMP integer program (PMPIP) and exploit this solution to find low-cost feasible solutions to the PMPIP. We will call this relaxation the PMPLP. In particular, we are interested in the *support graphs* of the relaxations. In our case, since we posed the PMP over a complete graph, where the nodes are demand/median locations, the *support graph* of the PMPLP solution contains the nodes of the complete graph together with all edges whose corresponding variables take on positive value in the solution of the PMPLP. All the edges whose corresponding variables have a value of 0 in the PMPLP solution are not in the support graph.

Note that in considering the support graph of the PMPLP, we simply use the edges whose corresponding variables are positive in the PMPLP solution. We do not create any new edges by shortcutting shortest paths (shortcutting means adding a new edge between two nodes with cost equal to the cost of the minimum cost path between the two nodes). Thus, if an edge variable is not positive in the PMPLP, then that edge is not in the PMPLP support graph, and the two ends of that edge cannot be linked in a PMP solution contained in the PMPLP support graph. Since many of the edges present in the complete graph may not be present in the PMPLP support graph, it is possible that for every potential subset of nodes, M, with $|M| = p$, there exists a node, $v \in V \backslash M$, such that none of the edges directly connecting v to a node in M are present in the PMPLP support graph. That is, it is possible that no dominating set of size p exists in the PMPLP support graph. In this case, for any choice of p medians, there will be some node that cannot be linked to a median using an edge of the PMPLP support graph. Thus, the support graph of the PMPLP will not contain a solution of the PMP, and some alteration of the PMPLP will be required. We explore three ways to alter the PMPLP in §3.1.

In certain cases, for example when the PMP instance is too large to be solved by current linear programming software or if the linear program is too slow, it is necessary to solve the PMP using heuristics. Although these heuristic solutions may not be optimal, we expect that a combination of these solutions will contain information that will help us build a better solution than any of the individual solutions. In a similar manner to the PMPLP, we can build a support graph out of multiple heuristics by simply including any edge of the complete graph that is used in at least one of the heuristic solutions.

Whether we use the PMPLP or heuristics to create a support graph, we will use dynamic programming to mine the useful information contained in it. While Hribar and Daskin (1997) have previously developed a dynamic programming heuristic for the PMP, our method is unique because we use branch decompositions to form the dynamic programming subproblems. Branch decompositions were introduced by Robertson and Seymour (1991), and they have proved to be an efficient means of solving NP-hard problems on certain graphs. For example, Cook and Seymour (2003) used branch decompositions in a heuristic for the traveling salesman problem, and Hicks (2004) used them to solve the graph minor containment problem.

Branch decompositions are essentially an assignment of the edges of a graph to the leaves of a binary tree. For the purposes of dynamic programming, they provide a hierarchical decomposition of a graph into subgraphs that will form the subproblems of the dynamic programming algorithm. Figure 1 gives an example of a graph together with a branch decomposition of the graph. We now give a formal definition of a branch decomposition.

Definition 2. Consider a graph, $G = (V, E)$, and a tree, T, and denote the leaves of T by L. Suppose that each vertex of $T \backslash L$ has degree exactly 3, and suppose further there exists a bijection, $\tau : E \leftrightarrow L$. Then we say that the pair (T, τ) is a *branch decomposition* of G

We will use the branch decomposition to define the subproblems in a dynamic programming scheme. Towards this end, throughout the rest of this paper we will assume that the branch decompositions are rooted. We root our branch decompositions by picking some

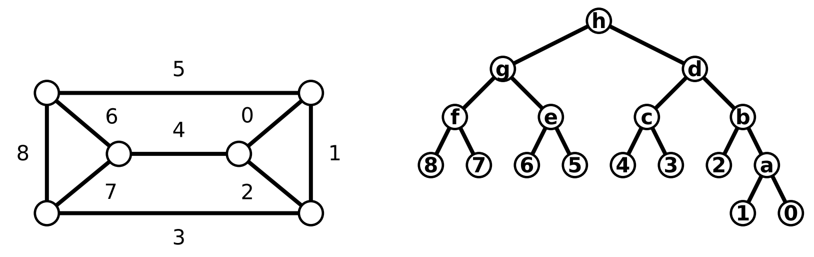

Figure 1 An example of a branch decomposition.

Note. Each of the edges in the support graph (left graph) is assigned to one of the leaves of the the branch decomposition (right graph). Also, each interior vertex except the root (labeled h) has degree exactly 3.

arbitrary edge of the tree and subdividing it. The new vertex of degree 2 that results from subdividing the edge will be the root of the tree.

The ideal branch decomposition for dynamic programming would be such that the edges assigned to different branches of the branch decomposition tree would have little interaction with each other in the graph, i.e. the edges would not be incident to the same node. The set of nodes of G that have incident edges from two different branches of a branch decomposition is called the *middle set*. We want to use branch decompositions with small middle sets. This idea is formalized in the concept of *branchwidth*.

Definition 3. Consider a graph, G, with a rooted branch decomposition, T, of G. Given a vertex, v, of T, let S denote the graph induced by the edges corresponding to leaves descending from v. The set of nodes of G that are incident to both edges in S and edges in $G\backslash S$ is called the *middle set* of v.

Definition 4. Consider a graph, G, with the set B of all rooted branch decompositions of G. For each branch decomposition $b \in B$, let s_b be the size of the largest middle set of b. Then the *branchwidth* of G is the minimum s_b, over all $b \in B$.

3. The Branch Decomposition Heuristic

Dynamic programming algorithms have two essential components: a decomposition of the problem into subproblems and a method for using smaller subproblem solutions to build solutions for larger subproblems. In this section, we describe how we use branch decompositions to build these components and incorporate them into our algorithm.

In our algorithm, we consider the directionality of the edges when we solve the smallest subproblems in the dynamic programming. Thus, rather than using branch decompositions of directed graphs we only consider the branch decompositions of the underlying undirected graphs. In theory, this distinction makes little difference because, given a directed graph, the branch decomposition of the underlying undirected graph can be extended to a branch decomposition of the directed graph by simply adding a pair of leaves representing each direction descending from the leaf representing the undirected edge. This extension does not affect the branchwidth of the decomposition.

For our heuristic, given a support graph, G, that comes from either linear programming or heuristics, and a branch decomposition of G, we start by defining the subproblems for dynamic programming. We have a subproblem for each vertex of the branch decomposition tree, and for a given tree vertex, the corresponding subproblem is a p-median problem on the subgraph induced by the leaves descending from that tree vertex. For leaf vertices of the branch decomposition tree, these subproblems are p-median problems on single edges of G. For our algorithm, we must determine how the allowed p medians will be distributed on the subgraphs. It is likely that, in an optimal solution, the medians will be spread throughout the graph, and the subgraphs will only use a fraction of the medians. However, we know of no theory that will allow us to distribute the medians to the branches of the decomposition while guaranteeing that solution quality does not suffer. Therefore, we simply allow each branch of the decomposition to use the full allotment of p medians. We ensure that no more than p medians are used in the final solution by only merging partial solutions that together have no more than p medians.

Since all p medians are allowed on each branch, there are six possible solutions to the subproblems on the leaves of the decomposition, see Figure 2. Each end of the corresponding edge can either be a median, be linked to the other end of the edge, or it can be free. Free nodes are necessary because, in a solution of the full problem, it is possible that a node may be linked to a median that is not part of the current subproblem, and if a node is either a median or linked to a median in a subproblem it will never be linked to a median outside of the subproblem. However, if one of the nodes incident to the edge corresponding to a leaf vertex has a degree of 1 in G, then that node will not be part of the middle set of any branch decomposition vertex and cannot be free. For such edges, there are at most the 4 possible solutions where the node with degree 1 is not free.

Once we have solutions for the leaves of the decomposition, we form solutions to the interior vertex subproblems by merging a partial solution from one of the children of the interior vertex with a partial solution from the other child of the vertex. The operation of

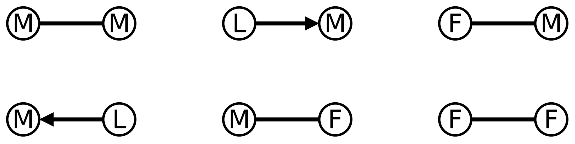

Figure 2 Dynamic programming base cases for leaves of decomposition.

Note. M denotes a median node, L denotes a node linked to a median, and F denotes a free node.

Table 1 Compatibility for middle set nodes

	Median	Linked	Free
Median	Compatible	Incompatible	Incompatible
Linked	Incompatible	Incompatible	Compatible
Free	Incompatible	Compatible	Compatible if in middle set

merging solutions forms the solution of the new subproblem by scanning through each of the child subproblems and setting each node and edge in the new subproblem to have the same value as in one of the child subproblems. Thus, a node that is linked to a median in a child subproblem remains linked, a node that is a median remains a median, etc. Nodes that are not in either child subproblem are also not part of the subproblem on the new vertex. It is obvious that this process leads to conflicts if the two child subproblems disagree on the state of a node, i.e. one child has a node linked to certain median and the other child has the node linked to a different median. We call these partial solutions *incompatible.*

Our goal with defining compatible solutions is to reduce the number of partial solutions that we need to merge in the dynamic programming stage. Since we do not need to merge an incompatible pair of solutions, it is in our best interest to make as many pairs incompatible as is possible while still retaining an optimal solution. We say that two states of a middle set node are compatible if a partial solution with one state at the node can be merged with a partial solution that has the other (not necessarily different) state on that node without violating some property of integral p-median solutions. We say that two partial solutions are compatible if each of the nodes in the intersection of their middle sets have compatible states. The compatibilities of the three states for middle set nodes are given in Table 1.

Most of the definitions in Table 1 are straightforward; however, we will still take the time to explain them here. A median state is compatible with a median state because both partial solutions agree on the state of the node, thus the solutions can be merged without a problem. However, the median state is not compatible with the linked state because a merged solution would have that node being both a median and linked to another median, which is unnecessary because a median node is considered to satisfy its own demand. Median nodes are also defined to be incompatible with free nodes. We chose to make these states incompatible to reduce the number of compatible solutions. This definition is safe because if a middle set node is a median in one partial solution (L) and free in the other (R), then it will be a median in the merged solution. Thus, we get the same merged solution as we would have by merging L with the partial solution (P) obtained by changing the node to a median in R. P must be one of the partial solutions of the R child branch because if L and R could be merged without exceeding p medians, then there are no more than p medians in P. Thus, P must be in the list of partial solutions for the R child branch. It follows that we are not discarding any possible solutions by declaring the median state to be incompatible with the free state.

We saw that the linked state was incompatible with the median state. The linked state is also incompatible with the linked state. If a middle set node is linked to a node that is not in the middle set in one partial solutions, then in the merged solution, it will be linked to two different nodes, which is unnecessary because linking to one median is all that is needed to satisfy a node's demand. If the node is linked to the same middle set node in both partial solutions, then we could have just used one partial solution where the node was free. Thus, the linked state is incompatible with the linked state. However, the linked state is compatible with the free state, because in that case the node is linked to a single median in the merged solution.

The only pair left to determine is the compatibility of the free state with the free state. If the node is not just in the middle set of the child branches but is also in the middle set of our current branch decomposition vertex, then we must have that the free state is compatible with the free state. However, nodes cannot be free in the final solution. So, if the node is in the middle set of the child vertices but not in the middle set of our current branch decomposition vertex, then the free state is not compatible with the free state.

Now that we have a rule for determining compatibility, we build partial solutions for a branch decomposition vertex by merging compatible partial solution pairs from its two children. However, when building solutions in this manner, it is possible that two partial solutions, A and B, will have the same number of medians and the same middle set configuration, but different configurations on the nodes that are not in the middle set. A and B will both be compatible with the exact same set of solutions from the other branch decomposition vertices. However, if A costs less than B, then A will give the lower cost in the merged solutions. Thus, we only need to store the lowest cost partial solution with each middle set configuration and number of medians. We store the partial solutions in a hash table with the hashing function

$$f(x_0, x_1, \ldots, x_{BW-1}, m) = \sum_{i=0}^{BW-1} 3^i x_i + 3^{BW} m$$

where $x_i \in \{0, 1, 2\}$ represent the state of the middle set node i, BW is the branchwidth, and m is the number of medians used.

Since we are building solutions by merging partial solutions from the children, we must first obtain the partial solutions on the children. A post-depth-first search ordering ensures that the children will be processed before the parent. When we finally reach the root vertex of the branch decomposition, the partial solutions become complete, and we choose the lowest cost solution at the root vertex to be the solution returned by our algorithm. The dynamic programming procedure is summarized in Algorithm 1.

3.1. Performance Tweaks

In our numerical experiments, and in practical application, situations arise where Algorithm 1 needs to be tweaked somewhat. First, as explained in §2, it is possible that no feasible solution exists for the given p on the PMPLP support graph. Since our branch decomposition algorithm on the PMPLP support (BDPM-LP) finds solutions only on the PMPLP support graph, our algorithm may not be able to find a feasible solution. There are three ways to deal with this problem. The first way is to shortcut the graph. This method essentially entails adding edges to the support graph until a feasible solution can be found. Unfortunately, adding edges to the graph can increase it's branchwidth, and since the complexity of our algorithm is so closely tied to branchwidth, shortcutting will likely have a detrimental effect on efficiency.

The second way to deal with the problem, is to add a cutting plane to the PMPLP. If no integral solution exists on the support graph of the PMPLP, then any integral solution must contain an edge that is not in the support graph. Let E be the set of edges in the support graph of the PMPLP. Then the constraint

$$\sum_{(i,j)\notin E} x_{i,j} \geq 1$$

Algorithm 1 Dynamic Programming on Branch Decompositions

```
Given a post-DFS ordering of branch decomposition
Initialize leaves of branch decomposition with their possible solutions
for branch decomposition vertices in post-DFS order do
    for i a solution of left child do
        for j a solution of right child do
            if (i and j are compatible) then
                Store the compatible pair (i,j)
            end if
        end for
    end for
    for each compatible pair (i,j) do
        merge i and j to get new solution
        if (cost of new solution < cost of solution currently stored in hash table) then
            store new solution in hash table
        end if
    end for
end for
Output least cost solution of root vertex
```

is a valid cutting plane for the PMPIP. Adding these cutting planes will eventually force an integral solution to exist on the support graph of the linear program. Unfortunately, this cutting plane does not seem to be facet-inducing. In our preliminary experiments, this method did not perform well because each cutting plane did not cut off enough of the feasible region. Also, this method tended to increase the branch-width of the support graph.

The third method, the one that we use in our computational experiments, is what we will call fixing. *Fixing* entails picking some node variable, $x_{i,i}$, that takes positive, fractional weight in the PMPLP, and adding the constraint $x_{i,i} = 1$. The PMPLP is then resolved with this added constraint, and we run our algorithm on the support graph of this new solution. Although this approach prevents the method from finding solutions where the fixed node is not a median, it does eventually force a feasible solution to exist on the PMPLP support graph.

Because the performance of BDPM-LP depends on the branchwidth of the PMPLP support, we want to avoid runnning the heuristic on graphs with high branchwidth. Fixing a single variable will not necessarily decrease the branchwidth and may in fact increase it. However, repeatedly fixing variables will eventually reduce the branchwidth of the support graph. This statement can easily be seen from the fact that fixing will eventually cause the PMPLP solution to be integral. Consequently, we can limit the branchwidths of the support graphs that we use for BDPM-LP by fixing whenever branchwidth is above the given limit. In our numerical experiments, our heuristic became impractical with branchwidths larger than 7. Consequently, we tweaked BDPM-LP by fixing whenever branchwidth is above 7. In a similar manner, the algorithm using multiple heuristic runs (BDPM-H) can have branchwidths that are too high. We correct this by using fewer heuristics to create the support graph.

There are multiple fixing strategies, such as fixing lexicographically or fixing the node with largest fractional weight. In our numerical experiments, we use lexicographic fixing. Given some indexing of the nodes, we simply scan the list of node variable values from the LP solution and fix the fractional node with the lowest index.

4. Complexity and Error Bounds

In this section, we give a complexity result and a theoretical error bound for BDPM-LP. We do not give similar error bounds for the version of our algorithm where the support graph comes from heuristics (BDPM-H) because the heuristics themselves generally do not have error bounds. In general, all that we can say about the error of BDPM-H is that it is at least as good as the heuristics used to build the support graph.

THEOREM 1. *Let BW be the branchwidth of a branch decomposition of a graph with N vertices and M edges. Then, Algorithm 1 using the given decomposition requires* $\mathcal{O}(m\log(m)(N + 9^{BW}BW + 5^{BW}N))$ *operations.*

Proof: The branch decomposition has $m\log(m)$ vertices, and at each vertex the algorithm needs to find the pairs of compatible solutions and merge them. To do this, we first need to find the middle set of the branch decomposition vertex. This step requires checking the induced degree of each node against its actual degree. Thus, the step requires N

comparisons. Now, each node in a middle set can have at most three configurations. Thus, each vertex of the decomposition has at most 3^{BW} partial solutions.

To find the compatible solutions, the algorithm compares the configuration of each node in the middle set for each pair of partial solutions, one from each child vertex in the decomposition. Since the number of nodes in the middle set is limited by BW, this step uses at most $3^{BW}3^{BW}BW = 9^{BW}BW$ comparisons. Finally, the compatible solutions must be merged. Since all configurations are not compatible with each other, we have a maximum of 5^{BW} compatible pairs. The solutions are merged by simply scanning through each of the partial solutions and setting the new solution to its values in the partial solutions. Since there are at most N nodes that must be scanned, merging the solutions requires $5^{BW}N$ steps. Thus, the complexity of the algorithm is $\mathcal{O}(m\log(m)(N + 9^{BW}BW + 5^{BW}N))$. $\square$

The fixing strategy introduced in §3.1 complicates the development of theoretical worst-case error bounds for BDPM-LP because the optimal cost of the linear program increases when fixing occurs, but if the fixing step is not necessary, then we can provide a theoretical bound.

THEOREM 2. *Let $\bar{x}$ be the optimal solution to the PMPLP, and let α be the smallest integer such that $\alpha\bar{x}$ is integral. Let $\bar{h}$ be the solution returned by algorithm 1, and let c be the cost vector for the instance. If no fixing is required, then $c^T\bar{h} \leq \alpha c^T\bar{x}$.*

Proof: Since $\alpha\bar{x}$ is integral, it contains at least one copy of each edge in the PMPLP support graph. Algorithm 1 returns a solutions whose edges are limited to the PMPLP support, and this solution has at most one copy of each of these edges. So, $\alpha\bar{x}$ has at least as many copies of each edge as are contained in $\bar{h}$. Thus, $c^T\bar{h} \leq \alpha c^T\bar{x}$. $\square$

For the interesting case of half-integral solutions, theorem 2 gives an error bound of 2 when fixing is not needed. Note that on these limited instances, our bound is better than the $1 + \sqrt{3} + \epsilon$ bound of Li and Svensson (2013), but in general it is worse. However, in our computational experiments, whether fixing was needed or not, the average performance of our algorithm is much better than the worst-case bound.

5. Computational Experiments

In this section, we report computational results for both the PMPLP-based algorithm and the heuristic-based algorithm. The first strategy for which we report computational results is the use of multiple runs of the GRASP heuristic of Feo and Resende (1995) to create the support graph. In our experiments, we use four runs of GRASP and we call this heuristic BDPM-GRASP. The second strategy for which we report results is BDPM-LP. We compare the results of our algorithm to GRASP, integer programming, the imp-GA algorithm of Rebreyend et al. (2015), and the hybrid heuristic (HHP) of Resende and Werneck (2004). The imp-GA algorithm was demonstrated by Rebreyend et al. (2015) to outperform other genetic algorithms on large problems, and HHP is also known to be an effective heuristic for p-median problems (see for example Avella et al. (2012)).

Our computational results were obtained on a Dell Precision T1650 workstation with a 3.3 GHz Intel Core i3-2120 CPU, 3.7 GB of RAM, and Red Hat Enterprise Linux version 6.6. The code was written in C++ and compiled with g++ version 4.4.7. Integer and linear programs were solved using Gurobi version 5.5.0 with the dual simplex method and a single thread. Branch decompositions were found using the C++ code of Hicks (2002). GRASP results and HHP results were obtained from the POPSTAR code of Resende and Werneck (2004). We obtained imp-GA results from our own implementation in C++.

The first set of instances that we test comes from the TSPLIB of Reinelt (1991). From this library, we used lin318, rd400, si535, ali535, rat575, gr666, u724, dsj1000, pr1002, rl1304, nrw1379, fl1400, u1432, vm1748, d2103, and pcb30308. Many of these instances have been used before as test instances for the p-median problem, see for example Avella et al. (2007) and García et al. (2011). For each file chosen from the TSPLIB, we ran instances with the number of medians, p, in the set 5, 10, 50, 100, 200, 300, 400, 500, up to half the number of vertices in the instance.

The second set of instances that we test are the OR-Library instances from Beasley (1985) . In keeping with standard practice in recent papers on p-median, see for example Avella et al. (2007), Elloumi (2010), and García et al. (2011), we only report computational results for instances 26-40.

The running times of both BDPM versions show a loose correlation with branchwidth, which should be expected given Theorem 3.1. However, since it is possible for the decomposition to have only one large middle set and for the rest of the middle sets to be small, or for a linear program solution to be mostly integer with only a small fractional part with high branchwidth, higher branchwidths do not necessarily lead to higher running times.

For both the TSPLIB instances and the OR-Library instances, BDPM-LP is generally faster and can solve larger problems than integer programming. It is also more accurate than HHP when fixing is not required. BDPM-LP was faster than imp-GA and more accurate than imp-GA when $p > 50$. If the time to solve the linear program is not counted, then the running times of BDPM-LP were competitive with HHP for low branchwidths. Also, BDPM-GRASP was able to improve on the best GRASP run, was faster than imp-GA, and was more accurate than imp-GA for $p > 100$. The following subsections give more detailed discussion of these results.

5.1. BDPM-GRASP

For BDPM-GRASP, we use four runs of GRASP to form the support graph for our algorithm. For some instances, the branchwidth of the support graph using four runs was too high for BDPM-GRASP to be practical. For these instances, we simply decrease the number of GRASP runs that we use to build the support graph. Detailed results for these instances, including the number of runs required, are reported in Table 9. We compare the results of BDPM-GRASP to integer programming, imp-GA, and to the best GRASP run from the four used to build the support graph. Average results for each of these methods are reported in Table 2. The averaged results in this table show that BDPM-GRASP slightly reduced the error of the best GRASP run, and BDPM-GRASP only had smaller error than imp-GA on the ORLIB instances. However, the averaging hides a clear trend in our data. Our data shows that BDPM-GRASP performs better as p increases, and imp-GA performs worse. We consistently saw improvement over the best GRASP run when p was at least 50. Likewise, BDPM-GRASP consistently had less error than imp-GA when p was at least 200 for small instances and 100 for large instances. These trends are seen in Tables 3 and 4.

For the OR-Library instances, the best GRASP run was often exact. However, on average over all the OR-library instances, the error ratio (calculated as the average of the heuristic cost divided by the true solution cost for each instance) of BDPM-GRASP was 1.00018 (this error ratio was calculated using the best GRASP solution for the two instances pmed37 and pmed40 where branchwidth was too high for BDPM-GRASP), while the best of the GRASP runs had an average error ratio of 1.00153. Imp-GA had an average error ratio of 1.00136. On average BDPM-GRASP removed 32.9% of the error of the best GRASP run (calculated as the average of the percentage of the GRASP error that was still present in BDPM-GRASP for each instance). Thus, on these instances, BDPM-GRASP was more accurate than both imp-GA and GRASP. Detailed results for these instances are reported in Table 5.

For the TSPLIB instances, we divide the instances into two groups. The small instances have less than 1000 vertices, and the large instances have at least 1000 vertices. The average error ratio of our algorithm for small instances was 1.0373 compared to an average error ratio of 1.0392 for the best GRASP run and 1.004 for imp-GA. On average for small instances, BDPM-GRASP removed 27.8% of the error of the best GRASP run. Detailed results for small instances are reported in Table 6. The average error ratio of BDPM-GRASP for large instances was 1.0353 compared to an average error ratio of 1.0388 for the best GRASP run. On average for large instances, BDPM-GRASP removed 37.0% of the error of the best GRASP run. Detailed results for large instances are reported in Tables 7 and 8.

Although the average results show a small improvement from using BDPM-GRASP over GRASP, and no improvement from using BDPM-GRASP over imp-GA, the benefit of using BDPM-GRASP becomes apparent when results are broken down according to the p values in the instances. Tables 3 and 4 and Figure 3 show average relative errors for different p values, and it is clear from Figure 3 that as p increases, the average BDPM-GRASP solution improves relative to the GRASP and imp-GA solutions. Tables 3 and 4 indicate that BDPM-GRASP offers significant improvement over GRASP when p is at least 50. For the small instances, Table 3 shows that BDPM-GRASP outperforms imp-GA when p is 200 or 300, and for the large instances, Table 4 shows that BDPM-GRASP outperforms imp-GA when p is at least 100.

Table 2 Average results for BDPM-GRASP

Algorithm	ORLIB			Small TSP			Large TSP		
	# solved	time	error	# solved	time	error	# solved	time	error
BDPM-GRASP	13	45.3	1.0002	36	22.19	1.037	64	60.6	1.035
GRASP	15	0.03	1.0015	36	0.025	1.039	64	0.13	1.039
imp-GA	15	730	1.0014	36	551.3	1.004	64	5588	1.023
IP	12	998.6	1.0	36	270.6	1.0	29	162.1	1.0

The # solved column indicates the number of instances that the relevant algorithm solved without exhausting available memory. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required to run GRASP.

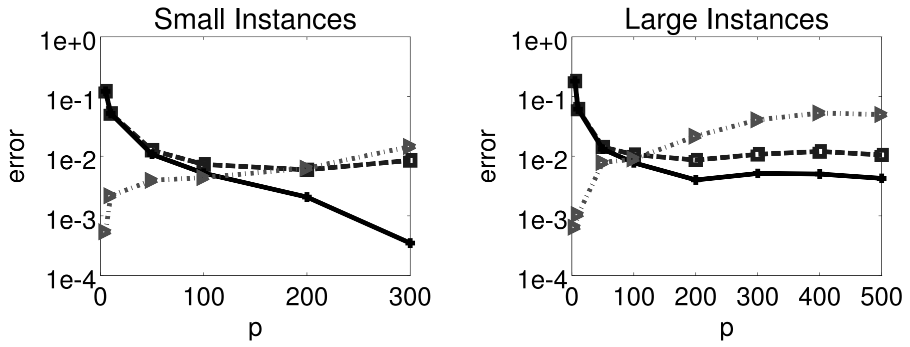

Figure 3 Comparison of BDPM-GRASP to GRASP and imp-GA for TSPLIB instances.

Note. The left plot compares average relative error for BDPM-GRASP (solid line, +), the best GRASP run (dashed line, □), and imp-GA (dash-dotted line, ▶) for different values of p and test instances from the TSPLIB with less than 1000 vertices. The right plot makes the same comparison for test instances with at least 1000 vertices.

Table 3 Average relative errors for BDPM-GRASP on small TSPLIB instances broken down by p

Algorithm	p=5	p=10	p=50	p=100	p=200	p=300
BDPM-GRASP	0.122	0.052	0.011	0.0052	0.0021	0.00035
GRASP	0.12	0.052	0.013	0.0073	0.0059	0.0085
imp-GA	0.00054	0.0022	0.0039	0.0044	0.0063	0.015

Note that BDPM-GRASP performs better as p increases, while imp-GA performs worse as p increases.

Table 4 Average relative errors for BDPM-GRASP on large TSPLIB instances broken down by p

Algorithm	p=5	p=10	p=50	p=100	p=200	p=300	p=400	p=500
BDPM-GRASP	0.1813	0.0615	0.0131	0.0078	0.004	0.0051	0.0050	0.0043
GRASP	0.1813	0.0615	0.0143	0.0106	0.0086	0.0108	0.0120	0.0105
imp-GA	0.0006	0.0011	0.0078	0.0090	0.0216	0.0407	0.0529	0.0498

Note that BDPM-GRASP performs better as p increases, while imp-GA performs worse as p increases.

Table 5 Results for OR-Library instances using branch decompositions of four GRASP heuristic runs.

Instance Data			IP Results		BDPM-GRASP			Best GRASP		imp-GA	
Name	size	p	cost	time	BW	time	error	time	error	time	error
pmed26	600	5	9917	1334	5	0.55	1.000	**0.04**	**1.000**	110	1.000
pmed27	600	10	8307	672	4	0.14	1.000	**0.03**	**1.000**	194	1.000
pmed28	600	60	4498	8.8	14	K	N/A	**0.01**	**1.0004**	692	1.0009
pmed29	600	120	3033	6.9	4	**1.77**	**1.000**	0.01	1.002	912	1.0046
pmed30	600	200	1989	7.1	3	**0.25**	**1.000**	0.01	1.009	921	1.0065
pmed31	700	5	10086	1687	4	0.62	1.000	**0.05**	**1.000**	128	1.000
pmed32	700	10	9297	1678	6	0.36	1.000	**0.04**	**1.000**	268	1.000
pmed33	700	70	4700	13.1	11	K	N/A	0.01	1.001	**1085**	**1.000**
pmed34	700	140	3013	12.1	8	K	N/A	0.01	1.006	**1522**	**1.0033**
pmed35	800	5	K	K	3	0.06	1.000	**0.05**	**1.000**	168	1.000
pmed36	800	10	K	K	15	K	N/A	**0.05**	**1.000**	350	1.000
pmed37	800	80	5057	18.8	16	K	N/A	**0.01**	**1.000**	1652	1.0024
pmed38	900	5	11060	6520	4	0.07	1.000	**0.08**	**1.000**	290	1.000
pmed39	900	10	K	K	6	0.13	1.000	**0.05**	**1.000**	381	1.000
pmed40	900	90	5128	26.1	22	K	N/A	**0.02**	**1.002**	2274	1.0027

Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required to run GRASP. Thus, the total time required for BDPM-GRASP is the reported time plus four times the time reported in the Best GRASP column. K indicates that the algorithm ran out of memory. BW stands for branchwidth. The column labeled Best GRASP gives the result of the best GRASP run. Optimal solutions specified in the OR-Library data set were used to calculate error ratios.

Table 6 Results for small TSPLIB instances using branch decompositions of four GRASP heuristic runs.

Instance Data		IP Results		BDPM-GRASP			Best GRASP		imp-GA	
Name	p	cost	time	BW	time	error	time	error	time	error
lin318	5	179778	6.08	3	0.18	1.173	0.01	1.173	**38**	**1.00002**
lin318	10	109398	2.73	1	0.01	1.047	0.01	1.047	**58**	**1.00005**
lin318	50	40350	3.42	5	0.54	1.010	0.01	1.014	**226**	**1.001**
lin318	100	18959	1.61	3	0.05	1.008	0.01	1.010	**186**	**1.0024**
rd400	5	68071	13.3	2	0.04	1.118	0.05	1.118	**70**	**1.0004**
rd400	10	46082	9.35	6	0.21	1.048	0.02	1.048	**133**	**1.0031**
rd400	50	17295	2.87	4	0.34	1.010	0.01	1.011	**273**	**1.0014**
rd400	100	10108	2.74	2	0.04	1.007	0.01	1.007	**336**	**1.002**
rd400	200	4532	1.93	2	0.08	1.007	0.01	1.012	**293**	**1.004**
si535	5	81303	11.6	1	0.02	1.042	0.04	1.042	**108**	**1.000**
si535	10	69532	13.4	2	0.04	1.022	0.02	1.022	**211**	**1.000**
si535	50	47195	4.29	5	0.33	1.002	0.01	1.004	**469**	**1.0019**
si535	100	38281	3.49	4	**0.18**	**1.0003**	0.01	1.0009	691	1.0021
si535	200	26992	3.50	3	**0.04**	**1.000**	0.01	1.0005	619	1.0002
ali535	5	965580	16.2	1	0.01	1.132	0.07	1.132	**98**	**1.000**
ali535	10	629984	15.8	6	3.40	1.080	0.03	1.080	**237**	**1.0006**
ali535	50	232100	7.60	6	9.28	1.014	0.01	1.017	**481**	**1.0031**
ali535	100	130639	5.92	3	0.32	1.007	0.01	1.008	**759**	**1.0047**
ali535	200	53127	5.53	3	**0.07**	**1.002**	0.01	1.005	687	1.0052
rat575	5	34971	2201	5	0.72	1.116	0.05	1.116	**178**	**1.0032**
rat575	10	23637	36.1	9	171	1.051	0.04	1.052	**324**	**1.0012**
rat575	50	9860	685	13	K	N/A	0.01	1.021	**659**	**1.0131**
rat575	100	6351	14.0	7	**413**	**1.008**	0.01	1.015	801	1.0113
rat575	200	3690	6.73	2	**0.05**	**1.003**	0.01	1.011	904	1.0079
gr666	5	1491343	36.8	1	0.02	1.128	0.09	1.128	**202**	**1.000**
gr666	10	993899	28.8	5	0.36	1.065	0.09	1.065	**328**	**1.0096**
gr666	50	401015	14.0	6	31.4	1.007	0.01	1.007	**795**	**1.0039**
gr666	100	250318	23.3	7	**80.7**	**1.002**	0.01	1.005	1178	1.0053
gr666	200	132376	10.4	3	**0.09**	**1.0002**	0.01	1.002	1290	1.0079
gr666	300	75917	5.25	2	**0.22**	**1.0007**	0.01	1.005	1036	1.0116
u724	5	268898	127	5	4.53	1.148	0.12	1.148	**243**	**1.0001**
u724	10	181564	6328	8	18.3	1.050	0.05	1.050	**457**	**1.0006**
u724	50	70291	51.9	9	57.8	1.011	0.01	1.014	**1109**	**1.0031**
u724	100	43949	12.9	4	0.74	1.004	0.01	1.005	**1486**	**1.0028**
u724	200	25756	9.33	3	**0.22**	**1.0002**	0.01	1.005	1580	1.0126
u724	300	16909	21.1	3	**0.23**	**1.000**	0.01	1.012	1303	1.0174

Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required to run GRASP. Thus, the total time required for BDPM-GRASP is the reported time plus four times the time reported for in the Best GRASP column. K indicates that the algorithm ran out of memory. BW stands for branchwidth. The column labeled Best GRASP gives the result of the best GRASP run. Where the PMPIP could not be solved, the PMPLP solution was used to calculate the error ratios.

Table 7 Results for large TSPLIB instances using branch decompositions of four GRASP heuristic runs.

Instance Data		IP Results		BDPM-GRASP			Best GRASP		imp-GA	
Name	p	cost	time	BW	time	error	time	error	time	error
dsj1000	5	142469699	157	1	0.06	1.134	0.23	1.134	**402**	**1.000002**
dsj1000	10	81510884	87.3	2	0.11	1.091	0.09	1.091	**731**	**1.00005**
dsj1000	50	34012410	71.7	5	8.40	1.011	0.02	1.012	**2253**	**1.0073**
dsj1000	100	22561557	65.5	6	**2.53**	**1.006**	0.02	1.009	3424	1.0076
dsj1000	200	13689404	28.7	3	**0.53**	**1.003**	0.02	1.005	3417	1.0140
dsj1000	300	9431896	30.6	4	**0.56**	**1.004**	0.03	1.006	3033	1.0347
dsj1000	400	6694084	10.5	2	**0.46**	**1.003**	0.03	1.007	2653	1.0431
dsj1000	500	4706364	17.2	2	**0.39**	**1.003**	0.03	1.008	2272	1.0490
pr1002	5	1923290	160	5	11.4	1.198	0.25	1.198	**430**	**1.00002**
pr1002	10	1263290	129	1	0.06	1.045	0.10	1.045	**810**	**1.00002**
pr1002	50	503512	85.7	6	12.4	1.012	0.02	1.012	**2187**	**1.0040**
pr1002	100	331435	33.3	6	**7.38**	**1.003**	0.02	1.008	3256	1.0060
pr1002	200	200172	43.4	4	**0.49**	**1.001**	0.02	1.006	3435	1.0179
pr1002	300	139232	17.0	3	**0.55**	**1.0001**	0.03	1.007	3053	1.0388
pr1002	400	104068	36.8	4	**0.57**	**1.001**	0.03	1.009	2667	1.0364
pr1002	500	78383	18.5	2	**0.52**	**1.001**	0.03	1.005	2287	1.0252
rl1304	5	3099632	633	7	42.0	1.137	0.57	1.137	**1066**	**1.00001**
rl1304	10	K	K	10	K	N/A	0.27	1.063	**1713**	**1.0012**
rl1304	50	795468	161	8	K	N/A	0.04	1.019	**4179**	**1.0058**
rl1304	100	491929	448	6	**17.3**	**1.008**	0.04	1.013	6048	1.0101
rl1304	200	268735	75.2	5	**1.48**	**1.004**	0.04	1.008	6186	1.0288
rl1304	300	177445	76.0	5	**1.03**	**1.003**	0.04	1.009	5672	1.0486
rl1304	400	128418	49.7	3	**1.22**	**1.002**	0.05	1.012	5139	1.0811
rl1304	500	97084	73.2	3	**2.48**	**1.002**	0.04	1.013	4675	1.0790
nrw1379	5	433349	384	2	0.14	1.124	0.50	1.124	**1106**	**1.0003**
nrw1379	10	K	K	9	29.2	1.052	0.22	1.052	**2021**	**1.0006**
nrw1379	50	K	K	15	K	N/A	0.07	1.015	**5406**	**1.0098**
nrw1379	100	K	K	14	K	N/A	0.04	1.016	**7597**	**1.0079**
nrw1379	200	K	K	8	K	N/A	**0.04**	**1.011**	7006	1.0251
nrw1379	300	K	K	4	**2.84**	**1.003**	0.04	1.009	6456	1.0393
nrw1379	400	K	K	3	**1.36**	**1.002**	0.05	1.006	5870	1.0462
nrw1379	500	K	K	4	**0.53**	**1.002**	0.06	1.006	5352	1.0481
fl1400	5	174844	311	2	0.13	1.468	0.26	1.468	**680**	**1.0040**
fl1400	10	100872	210	2	0.82	1.082	0.16	1.082	**1504**	**1.0035**
fl1400	50	28837	121	6	**22.6**	**1.009**	0.05	1.013	4320	1.0111
fl1400	100	K	K	8	**132**	**1.001**	0.04	1.005	6577	1.0124
fl1400	200	K	K	9	K	N/A	**0.04**	**1.006**	7243	1.0139
fl1400	300	K	K	10	K	N/A	**0.04**	**1.011**	6689	1.0371
fl1400	400	K	K	9	K	N/A	**0.05**	**1.020**	6086	1.0652
fl1400	500	K	K	9	K	N/A	**0.05**	**1.025**	5557	1.0671

Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required to run GRASP. Thus, the total time required for BDPM-GRASP is the reported time plus four times the time reported for in the Best GRASP column. K indicates that the algorithm ran out of memory. BW stands for branchwidth. The column labeled Best GRASP gives the result of the best GRASP run. Where the PMPIP could not be solved, the PMPLP solution was used to calculate the error ratios.

Table 8 **More results for large TSPLIB instances using branch decompositions of four GRASP heuristic runs.**

Instance Data		IP Results		BDPM-GRASP			Best GRASP		imp-GA	
Name	p	cost	time	BW	time	error	time	error	time	error
u1432	5	1210479	566	1	0.14	1.113	0.50	1.113	**1063**	**1.0001**
u1432	10	850078	600	7	28.4	1.054	0.22	1.054	**2197**	**1.0001**
u1432	50	K	K	11	K	N/A	0.04	1.015	**4809**	**1.0075**
u1432	100	K	K	9	K	N/A	0.04	1.013	**7454**	**1.0117**
u1432	200	K	K	8	K	N/A	**0.04**	**1.016**	7511	1.0239
u1432	300	K	K	12	K	N/A	**0.04**	**1.026**	6962	1.0401
u1432	400	K	K	11	K	N/A	**0.04**	**1.028**	6414	1.0400
u1432	500	K	K	9	K	N/A	**0.04**	**1.001**	5879	1.0040
vm1748	5	K	K	1	0.21	1.155	0.64	1.155	**1657**	**1.00002**
vm1748	10	K	K	7	20.0	1.057	0.34	1.057	**3083**	**1.0018**
vm1748	50	K	K	9	K	N/A	0.09	1.014	**7708**	**1.0055**
vm1748	100	K	K	7	825	1.007	0.05	1.009	**12091**	**1.0067**
vm1748	200	K	K	5	**4.63**	**1.002**	0.06	1.007	11628	1.0254
vm1748	300	K	K	6	**7.05**	**1.001**	0.06	1.007	10933	1.0511
vm1748	400	K	K	5	**4.92**	**1.0002**	0.07	1.006	10140	1.0727
vm1748	500	K	K	5	**3.00**	**1.001**	0.08	1.008	9477	1.0803
d2103	5	K	K	1	0.3	1.122	1.20	1.122	**2874**	**1.0006**
d2103	10	K	K	9	149	1.048	0.41	1.048	**3767**	**1.0012**
d2103	50	K	K	16	K	N/A	**0.11**	**1.014**	11033	1.0117
d2103	100	K	K	21	K	N/A	0.07	1.012	**17721**	**1.0095**
d2103	200	K	K	8	K	N/A	**0.06**	**1.010**	16859	1.0241
d2103	300	K	K	7	**62.7**	**1.003**	0.06	1.011	16149	1.0356
d2103	400	K	K	8	**24.0**	**1.004**	0.07	1.008	15344	1.0382
d2103	500	K	K	5	**8.69**	**1.007**	0.08	1.018	14450	1.0459

Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required to run GRASP. Thus, the total time required for BDPM-GRASP is the reported time plus four times the time reported for in the Best GRASP column. K indicates that the algorithm ran out of memory. BW stands for branchwidth. The column labeled Best GRASP gives the result of the best GRASP run. Where the PMPIP could not be solved, the PMPLP solution was used to calculate the error ratios.

Table 9 Results for instances using branch decompositions of less than four GRASP heuristic runs.

Instance Data			BDPM-GRASP					Best GRASP		imp-GA	
Name	size	p	runs	BW	cost	time	error	time	error	time	error
pmed28	600	60	2	6	4500	3.73	1.0004	**0.01**	**1.0004**	692	1.0009
pmed33	700	70	2	5	4700	**14.9**	**1.000**	0.01	1.001	1085	1.000
pmed34	700	140	3	4	3014	**2.74**	**1.0003**	0.01	1.006	1522	1.0033
pmed36	800	10	2	10	9934	564	1.000	**0.05**	**1.000**	350	1.000
pmed37	800	80	2	16	K	K	N/A	**0.01**	**1.000**	1652	1.0024
pmed40	900	90	2	8	K	K	N/A	**0.02**	**1.002**	2274	1.0027
rat575	575	50	2	5	10064	4.18	1.021	0.01	1.021	**659**	**1.0131**
rl1304	1304	10	3	7	2266366	16.9	1.063	0.27	1.063	**1713**	**1.0012**
rl1304	1304	50	3	6	807157	15.9	1.015	0.04	1.019	**4179**	**1.0058**
nrw1379	1379	50	2	4	132525	5.50	1.016	0.06	1.016	**5406**	**1.0098**
nrw1379	1379	100	2	5	89757	26.6	1.016	0.03	1.017	**7597**	**1.0079**
nrw1379	1379	200	3	6	58789	**287**	**1.006**	0.04	1.011	7006	1.0251
fl1400	1400	200	3	7	9205	**212**	**1.003**	0.04	1.006	7243	1.0139
fl1400	1400	300	3	7	6520	**427**	**1.008**	0.04	1.011	6689	1.0371
fl1400	1400	400	3	7	4953	**172**	**1.015**	0.04	1.020	6086	1.0652
fl1400	1400	500	3	7	3934	**110**	**1.018**	0.04	1.025	5557	1.0671
u1432	1432	50	2	5	367198	3.40	1.014	0.04	1.015	**4809**	**1.0075**
u1432	1432	100	2	4	246988	9.32	1.012	0.04	1.013	**7454**	**1.0117**
u1432	1432	200	3	7	161386	**201**	**1.009**	0.04	1.016	7511	1.0239
u1432	1432	300	2	3	125997	**3.41**	**1.019**	0.04	1.028	6962	1.0401
u1432	1432	400	3	7	104758	**144**	**1.013**	0.04	1.028	6414	1.0400
u1432	1432	500	2	3	93200	**2.66**	**1.000**	0.04	1.002	5879	1.0040
vm1748	1748	50	3	7	1018934	683	1.014	0.09	1.014	**7708**	**1.0055**
d2103	2103	50	2	5	306145	27.4	1.014	0.11	1.014	**11033**	**1.0117**
d2103	2103	100	2	5	196578	**68.8**	**1.009**	0.07	1.013	17721	1.0095
d2103	2103	200	3	8	118888	**23.3**	**1.004**	0.06	1.010	16859	1.0241
pcb3038	3038	10	3	7	1297532	216	1.0687	1.27	1.0687	**9929**	**1.000**
pcb3038	3038	50	2	7	K	K	N/A	0.24	1.0093	**27316**	**1.000**
pcb3038	3038	100	2	5	357653	138	1.0031	0.15	1.0037	**37658**	**1.000**
pcb3038	3038	200	2	5	240436	**173**	**1.000**	0.13	1.001	36296	1.0107
pcb3038	3038	300	2	4	189091	**58.3**	**1.000**	0.15	1.003	35052	1.0363
pcb3038	3038	400	2	4	157828	**13.6**	**1.000**	0.16	1.005	33699	1.0633
pcb3038	3038	500	3	7	135969	**57**	**1.000**	0.19	1.006	32542	1.0812

Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The time reported for BDPM-GRASP does not include the time required to run GRASP. Thus, the total time required for BDPM-GRASP is the reported time plus four times the time reported for in the Best GRASP column. K indicates that the algorithm ran out of memory. BW stands for branchwidth. The column labeled runs gives the number of GRASP runs used to build the support graph for BDPM-GRASP. The column labeled Best GRASP gives the result of the best GRASP run. For TSPLIB instances where the PMPIP could not be solved, the PMPLP solution was used to calculate the error ratios. For the pcb3038 instances the PMPLP could not be solved. So, we use the best available solution to calculate the error ratios.

5.2. BDPM-LP

The results of BDPM-LP show that our algorithm obtains high quality solutions and usually gets a better quality solution than both HHP and imp-GA. Table 10 gives average results for the different methods. The BDPM-LP algorithm does have a much higher computational cost than HHP since the PMPLP must be solved. However, when the linear program solution is already known, and the branchwidth of the support graph is at most 5, then BDPM-LP is faster than the HHP algorithm. Over all the TSPLIB instances, the average time required by the BDPM portion of BDPM-LP was 106 seconds compared to 4.8 seconds for the HHP algorithm. However, for the instances that did not require fixing, the average time required was 10.2 seconds compared to 3.6 seconds for the HHP algorithm. If we only consider instances that did not require fixing and had branchwidth at most 5, then the average time required by the BDPM portion of BDPM-LP was 1.2 seconds compared to 3.7 seconds for the HHP algorithm. So, if the root relaxation solution is already known and it has branchwidth at most 5, then BDPM-LP is actually faster than the HHP algorithm. The BDPM-LP method also outperforms the imp-GA method. In contrast to the results for BDPM-GRASP, Table 10 shows that BDPM-LP has less error than imp-GA even when results are averaged over all p values.

For most of the OR-Library instances, the HHP algorithm is exact, and the PMPLP solution is either integral or has high branchwidth. Table 13 lists the branchwidths for these instances. Because they either could not be solved or were solved by fixing without using the branch decomposition algorithm, we do not report results for these instances.

For the small (less than 1000 vertices) TSPLIB instances where the linear program solution was not integral, the average error ratio of BDPM-LP was 1.000783 compared to 1.0147 for the HHP algorithm and 1.0063 for imp-GA. On average for small instances, BDPM-LP had 79.2% less error than the HHP algorithm and 64.9% less error than imp-GA. Detailed results for these instances are reported in Table 14. Thus, BDPM-LP outperforms both HHP and imp-GA on these instances.

For the large (at least 1000 vertices) TSPLIB instances where the linear program solution was not integral, the average error ratio of BDPM-LP was 1.00238 compared to 1.00951 for the HHP algorithm and 1.0266 for imp-GA. On average for large instances, BDPM-LP had 33.3% less error than the HHP algorithm and 86.0% less error than imp-GA. Detailed results for these instances are reported in Table 15. Thus, BDPM-LP outperforms both HHP and imp-GA on these instances.

Much of the additional error for the large instances appears to be caused by fixing when the branchwidth of the linear program solution support graph was too high. If we remove the instances where we used fixing to reduce the branchwidth, then BDPM-LP had an average error ratio of 1.000413 compared to an average error ratio of 1.00656 for the HHP algorithm and 1.0290 for imp-GA. On average for these instances, BDPM-LP had 44.6% less error than the HHP algorithm and 96.4% less error than imp-GA.

As with the BDPM-GRASP results, we break the average results down by p values. Tables 11 and 12 and Figure 4 show average relative errors for different p values. Unlike BDPM-GRASP, BDPM-LP performs well for all p-values. For p values of 400 and 500, the linear program support graphs often had high branchwidth that was corrected with fixing. This fixing leads to higher errors for BDPM-LP for these p-values. However, Figure 4 shows that BDPM-LP still outperforms HHP for instances that did not require fixing to correct high branchwidths. BDPM-LP also outperforms imp-GA across almost all p-values, with imp-GA only having less error for the small instances with p values either 5 or 10.

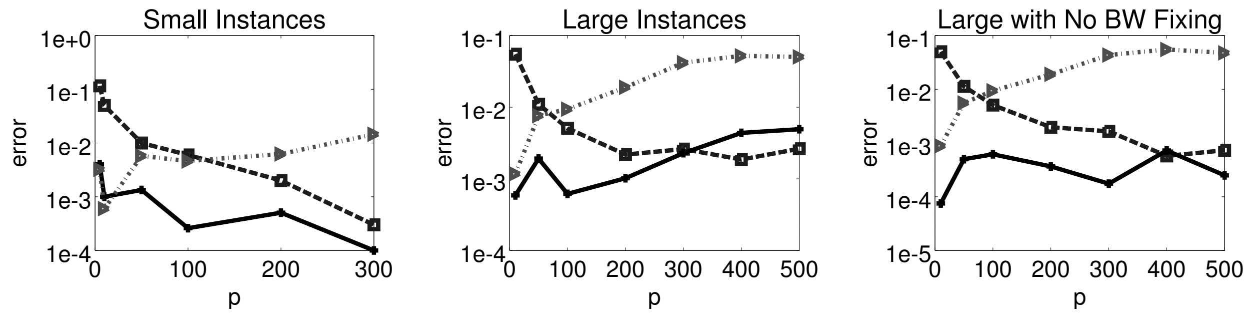

Figure 4 Comparison of BDPM-LP to HHP and imp-GA for TSPLIB instances.

Note. The left plot compares average relative error for BDPM-LP (solid line, +), HHP (dashed line, □), and imp-GA (dash-dotted line, ▶) for different values of p and test instances from the TSPLIB with less than 1000 vertices. The center plot makes the same comparison for test instances with at least 1000 vertices. The right plot makes the same comparison for instances with at least 1000 vertices that did not require fixing to correct high branchwidths.

Table 10 Average results for BDPM-LP

Algorithm	Small TSP			Large TSP		
	# solved	time	error	# solved	time	error
BDPM-LP	16	25.2	1.0008	39	493	1.0024
HHP	16	0.97	1.0147	39	6.38	1.0095
imp-GA	16	713	1.0063	39	5583	1.0266
IP	16	585	1.0	12	83	1.0

The # solved column indicates the number of instances that the relevant algorithm solved without exhausting available memory. Time is reported in seconds.

Table 11 Average relative errors for BDPM-LP on small TSPLIB instances broken down by p

Algorithm	p=5	p=10	p=50	p=100	p=200	p=300
BDPM-LP	0.004	0.001	0.0013	0.0003	0.0005	0.0001
HHP	0.116	0.05	0.01	0.006	0.0020	0.0003
imp-GA	0.0032	0.0006	0.0058	0.0046	0.0062	0.0145

Note that BDPM-LP performs better than HHP for all p values, and better than imp-GA for p at least 50.

Table 12 Average relative errors for BDPM-LP on large TSPLIB instances broken down by p

Algorithm	p=10	p=50	p=100	p=200	p=300	p=400	p=500
BDPM-LP	0.0006	0.0019	0.0006	0.0010	0.0023	0.0044	0.0049
HHP	0.055	0.011	0.0051	0.0022	0.0026	0.0019	0.0026
imp-GA	0.0012	0.0076	0.0094	0.0190	0.0418	0.0521	0.0504

Note that BDPM-LP performs better than HHP for p less than 400, and better than imp-GA for all p values.

Table 13 Branch-Widths of PMPLP Support Graphs for OR-Library Instances.

Instance Data			IP Results		
Name	size	p	cost	time	Branchwidth
pmed26	600	5	9917	1334.1	22
pmed27	600	10	8307	671.8	33
pmed28	600	60	4498	8.8	1
pmed29	600	120	3033	6.9	1
pmed30	600	200	1989	7.1	1
pmed31	700	5	10086	1686.8	7
pmed32	700	10	9297	1677.6	34
pmed33	700	70	4700	13.1	1
pmed34	700	140	3013	12.1	1
pmed35	800	5	K	K	10
pmed36	800	10	K	K	40
pmed37	800	80	5057	18.8	2
pmed38	900	5	11060	6520	23
pmed39	900	10	K	K	34
pmed40	900	90	5128	26.1	1

This table gives the branchwidths of the PMPLP support graphs for the instances of the OR-library. Most of these instances are either integral (indicated by a branchwidth of 1) or have branchwidth much higher than 7.

Table 14 Results for small TSPLIB instances using branch decompositions of the PMPLP support graph.

Instance Data		IP Results		BDPM-LP				HHP Algorithm		imp-GA	
Name	p	cost	time	BW	LP time	BD time	error	time	error	time	error
lin318	50	40350	3.42	2^F	1.02	0.53	1.002	0.20	1.010	**226**	**1.001**
lin318	100	18959	1.61	2	**0.76**	**0.04**	**1.000**	0.13	1.008	186	1.0024
rd400	100	10108	2.74	2	**1.39**	**0.07**	**1.000**	0.20	1.007	336	1.002
rd400	200	4532	1.93	2	**0.91**	**0.05**	**1.0007**	0.33	1.004	293	1.004
ali535	100	130639	5.92	2	**3.41**	**0.11**	**1.000**	0.53	1.007	759	1.0047
ali535	200	53127	5.53	3	**2.31**	**0.16**	**1.001**	0.53	1.002	687	1.0052
rat575	5	34971	2201	2^{BW}	62.5	12.6	1.004	2.54	1.116	**178**	**1.0032**
rat575	50	9860	685	7^{BW}	**17.3**	**93.3**	**1.002**	0.73	1.010	659	1.0131
rat575	100	6351	14.0	3^F	**9.83**	**1.85**	**1.0003**	0.57	1.006	801	1.0113
rat575	200	3690	6.73	2	**6.50**	**0.16**	**1.0003**	0.60	1.002	904	1.0079
gr666	100	250318	23.3	2^F	**8.72**	**0.74**	**1.001**	1.01	1.002	1178	1.0053
gr666	200	132376	10.4	2^F	**4.98**	**1.32**	**1.00002**	0.91	1.00002	1290	1.0079
gr666	300	75917	5.25	2	**2.75**	**0.10**	**1.000**	0.95	1.0002	1036	1.0116
u724	10	181564	6328	1^F	93.6	14.8	1.001	3.80	1.050	**457**	**1.0006**
u724	50	70291	51.9	6	**24.5**	**27.6**	**1.00001**	1.17	1.010	1109	1.0031
u724	300	16909	21.1	2	**9.48**	**0.15**	**1.0002**	1.39	1.0004	1303	1.0174

Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The LP time column gives the time required to solve the root relaxation. The BD time column gives the time required for BDPM-LP after the root relaxation was solved. K indicates that the algorithm ran out of memory. BW stands for branchwidth. If fixing was required, we indicate that with superscripts on the branchwidth. The BW superscript indicates that we fixed because of high branchwidths, and the F superscript indicates that we fixed because the PMPLP support graph did not contain a feasible solution. Where the PMPIP could not be solved, the PMPLP solution cost was used to calculate the error ratios. Instances for which the PMPLP solution was integral are not reported.

Table 15 **Results for large TSPLIB instances using branch decompositions of the PMPLP support graph.**

Instance Data		IP Results		BDPM-LP				HHP Algorithm		imp-GA	
Name	p	cost	time	BW	LP time	BD time	error	time	error	time	error
dsj1000	50	34012410	71.7	2^{F}	**53.2**	**3.36**	**1.001**	2.28	1.010	2253	1.0073
dsj1000	100	22561557	65.5	5	**39.1**	**0.83**	**1.0003**	2.06	1.006	3424	1.0076
dsj1000	200	13689404	28.7	2	**23.3**	**0.52**	**1.000**	1.75	1.003	3417	1.0140
dsj1000	300	9431896	30.6	2^{F}	**15.1**	**5.26**	**1.0002**	2.06	1.002	3033	1.0347
dsj1000	500	4706364	17.2	2	**6.4**	**0.26**	**1.0003**	2.54	1.002	2272	1.0490
pr1002	50	503512	85.7	2^{F}	**70.0**	**1.64**	**1.0003**	2.25	1.010	2187	1.0040
pr1002	200	200172	43.4	2^{F}	**18.9**	**3.85**	**1.0001**	1.98	1.0002	3435	1.0179
pr1002	400	104068	36.8	2^{F}	25.4	3.22	1.0003	**2.00**	**1.0002**	2667	1.0364
pr1002	500	78383	18.5	2^{F}	4.99	2.04	1.0005	**2.66**	**1.0001**	2287	1.0252
rl1304	10	K	K	1^{FBW}	1305	119	1.002	14.73	1.063	**1713**	**1.0012**
rl1304	100	491929	448	3	**109**	**0.75**	**1.0001**	4.05	1.007	6048	1.0101
rl1304	300	177445	76.0	2	**58.1**	**1.2**	**1.000**	3.17	1.003	5672	1.0486
rl1304	500	97084	73.2	2	**46.4**	**1.38**	**1.0001**	4.44	1.001	4675	1.0790
nrw1379	10	K	K	3	**714**	**0.73**	**1.00005**	15.43	1.052	2021	1.0006
nrw1379	50	K	K	1^{BW}	**437**	**460**	**1.004**	7.36	1.012	5406	1.0098
nrw1379	100	K	K	6	**109**	**157**	**1.001**	3.78	1.005	7597	1.0079
nrw1379	200	K	K	7	**183**	**45.1**	**1.001**	4.62	1.0027	7006	1.0251
nrw1379	300	K	K	2	**103**	**1.26**	**1.0003**	4.42	1.001	6456	1.0393
nrw1379	400	K	K	2^{F}	**67.3**	**19.1**	**1.0005**	4.28	1.001	5870	1.0462
nrw1379	500	K	K	2^{F}	**54.0**	**12.0**	**1.0003**	4.30	1.001	5352	1.0481
fl1400	100	K	K	5	**95.8**	**1.81**	**1.0003**	4.96	1.0005	6577	1.0124
fl1400	200	K	K	6^{BW}	145	169	1.003	**4.10**	**1.002**	7243	1.0139
fl1400	300	K	K	2^{FBW}	120	313	1.012	**5.66**	**1.005**	6689	1.0371
fl1400	400	K	K	5^{F}	89.4	20.8	1.002	**5.96**	**1.001**	6086	1.0652
fl1400	500	K	K	3^{FBW}	128	509	1.033	**6.54**	**1.014**	5557	1.0671
u1432	50	K	K	1^{FBW}	**273**	**193**	**1.003**	4.91	1.011	4809	1.0075
u1432	100	K	K	4^{F}	**149**	**38.4**	**1.001**	4.55	1.006	7454	1.0117
u1432	200	K	K	2^{FBW}	**121**	**56.7**	**1.001**	4.21	1.003	7511	1.0239
u1432	300	K	K	7^{BW}	**107**	**9.10**	**1.001**	5.79	1.004	6962	1.0401
u1432	400	K	K	5^{BW}	163	523	1.019	**6.36**	**1.007**	6414	1.0400
u1432	500	K	K	3	62.0	2.14	1.000	**4.20**	**1.000**	5879	1.0040
vm1748	10	K	K	4^{BW}	**1640**	**87.7**	**1.0002**	26.2	1.057	3083	1.0018
vm1748	50	K	K	3	**232**	**1.75**	**1.0002**	8.40	1.014	7708	1.0055
vm1748	100	K	K	5	**268**	**6.86**	**1.001**	6.00	1.006	12091	1.0067
vm1748	300	K	K	4	**118**	**2.52**	**1.0002**	7.22	1.0006	10933	1.0511
vm1748	400	K	K	2^{F}	80.7	4.56	1.0001	**6.88**	**1.0001**	10140	1.0727
vm1748	500	K	K	2	80.2	2.96	1.0003	**7.58**	**1.0003**	9477	1.0803
d2103	10	K	K	1^{F}	**2769**	**33.1**	**1.0001**	25.6	1.048	3767	1.0012
d2103	50	K	K	1^{BW}	**3396**	**2863**	**1.003**	13.7	1.009	11033	1.0117

Bold text indicates the method that had the least error, or the fastest method if the methods had the same error. Time is reported in seconds. The LP time column gives the time required to solve the root relaxation. The BD time column gives the time required for BDPM-LP after the root relaxation was solved. K indicates that the algorithm ran out of memory. BW stands for branchwidth. If fixing was required, we indicate that with superscripts on the branchwidth. The BW superscript indicates that we fixed because of high branchwidths, and the F superscript indicates that we fixed because the PMPLP support graph did not contain a feasible solution. Where the PMPIP could not be solved, the PMPLP solution cost was used to calculate the error ratios. Instances for which the PMPLP solution was integral are not reported.

6. Conclusions

In this paper, we introduced BDPM, a branch decomposition based dynamic programming algorithm for the p-median problem. This algorithm works both for finding a high quality solution in the linear programming support graph and for finding an improved solution out of a pool of heuristic solutions. However, BDPM is sensitive to branchwidth and branchwidths above 7 are not feasible on a typical desktop computer due to memory requirements. However, our computational experiments showed that BDPM can perform better than other state-of-the-art methods when these branchwidth restrictions are met.

The computational results in §5 show that our BDPM algorithm is a useful tool for improving on a pool of heuristic solutions. Using BDPM to improve a pool of heuristic solutions works best when p is sufficiently large. BDPM-GRASP significantly improved on GRASP solutions and also had less error than imp-GA for instances with p values of at least 100, while still being significantly faster than integer programming or our implementation of imp-GA. In terms of relative error, imp-GA was the better heuristic when p values were less than 100. The dependence of BDPM-GRASP on high p values corresponds to the observation that, as p increases, the difference between the GRASP runs used to form the support graph likely increases. Thus, the support graph likely has more edges and contains better solutions as p increases. A possible direction for future study is to investigate whether the use of different heuristics leads to a better pool on which to run BDPM.

The results in §5 also show that BDPM-LP is able to create high quality solutions when the branchwidth of the linear program support graph is at most 7. While slower than the HHP algorithm, BDPM-LP was more accurate. BDPM-LP was also faster and more accurate than our implementation of imp-GA. However, BDPM-LP requires a solution of the PMPLP, which may become difficult for larger problems. Higher branchwidths are not feasible on a typical desktop computer due to memory requirements, and although these higher branchwidths can be dealt with through fixing, such fixing led to higher errors in our experiments. Thus, a possible direction for future study is to investigate whether different fixing rules can lower the branchwidth of the support graph without hurting solution quality. Also, Theorem 2 gives an error bound on the result of BDPM-LP based on the smallest fractional part of the linear program support graph. This bound is not known to be tight and can likely be improved.

Acknowledgments

We would like to thank the anonymous reviewers whose comments improved the presentation of this paper. C. Fast was supported by NSF CMMI-1300477. I.V. Hicks was supported by the NSF CMMI-1300477 and CMMI-1404864.

References

Antamoshkin A, Kazakovtsev L (2013) Random search algorithm for the p-median problem. *Informatica (Slovenia)* 37(3):267–278.

Arya V, Garg N, Khandekar R, Meyerson A, Munagala K, Pandit V (2004) Local search heuristics for k-median and facility location problems. *SIAM J. Comput.* 33(3):544–562, ISSN 0097-5397.

Avella P, Boccia M, Salerno S, Vasilyev I (2012) An aggregation heuristic for large scale p-median problem. *Computers & Operations Research* 39(7):1625 – 1632, ISSN 0305-0548.

Avella P, Sassano A (2001) On the p-median polytope. *Math. Programming* 89(3):395–411, ISSN 0025-5610.

Avella P, Sassano A, Vasil'ev I (2007) Computational study of large-scale p-median problems. *Math. Program.* 109(1):89–114.

Beasley J (1985) A note on solving large p-median problems. *Eur. J. Oper. Res.* 21:270–273.

Cook W, Seymour PD (2003) Tour merging via branch-decomposition. *INFORMS J. Comput.* (15):233–248.

Elloumi S (2010) A tighter formulation of the p-median problem. *Journal of Combinatorial Optimization* 19(1):69–83, ISSN 1382-6905.

Feo T, Resende M (1995) Greedy randomized adaptive search procedures. *Journal of Global Optimization* 6:109–133.

García S, Labbé M, Marín A (2011) Solving large p-median problems with a radius formulation. *INFORMS J. Comput.* 23(4):546–556, ISSN 1526-5528.

Hicks IV (2002) Branchwidth heuristics. *Congressus Numerantium* 159:31–50.

Hicks IV (2004) Branch decompositions and minor containment. *Networks* 43(1):1–9, ISSN 1097-0037.

Hribar M, Daskin M (1997) A dynamic programming heuristic for the p-median problem. *Eur. J. Oper. Res.* 101(3):499 – 508, ISSN 0377-2217.

Kunkel A, Van Itallie E, Wu D (2014) Optimal distribution of medical backpacks and health surveillance assistants in malawi. *Health Care Management Science* 17(3):230–244, ISSN 1386-9620.

Li S, Svensson O (2013) Approximating k-median via pseudo-approximation. *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, 901–910, STOC '13 (New York, NY, USA: ACM), ISBN 978-1-4503-2029-0.

Rebreyend P, Lemarchand L, Euler R (2015) *A Computational Comparison of Different Algorithms for Very Large p-median Problems*, 13–24 (Cham: Springer International Publishing), ISBN 978-3-319-16468-7.

Reinelt G (1991) TSPLIB - A Traveling Salesman Problem Library. *INFORMS J. Comput.* 3:376–384.

Resende M, Werneck R (2004) A hybrid heuristic for the p-median problem. *J. Heuristics* 10(1):59–88.

Robertson N, Seymour PD (1991) Graph minors. x. obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B* 52(2):153 – 190, ISSN 0095-8956.

Rolland E, Schilling D, Current J (1997) An efficient tabu search procedure for the p-median problem. *Eur. J. Oper. Res.* 96(2):329 – 342, ISSN 0377-2217.