# Applying Well-Separated Pairs to the Travelling Salesman Problem

Jerry Qian
University of Maryland – CMSC828T
May 19, 2021
jerryq@umd.edu

**Abstract**

Well-separated pair decomposition offers geometric insights on distance closeness for sets of points in Euclidean space. We investigate the role of well-separated pairs in solving the travelling salesman problem. WSPs can improve existing TSP algorithms and be the basis for new ones. Additionally, we run experiments to find the pros and cons of using Point, PR, and PMR quadtrees when finding WSPDs for the travelling salesman problem.

## 1 Introduction

The travelling salesman problem (TSP) asks that given a list of points, what is the shortest path that visits each point once and returns to the origin point. Well-separated pairs (WSPs) are pairs of sets of points that are well-separated. Sets of points are considered well-separated when the distance between the two balls of radius $r$ surrounding either set is greater than $s * r$, where s is some separation factor [1].

WSPs have been used in a large variety of distance problems. The well-separated pair decomposition (WSPD) gives us geometric insights on the closeness between sets of points in Euclidean space. We apply WSPs to improve the performance of existing TSP algorithms and present a new polynomial-time algorithm.

## 2 Related Work

Optimal TSP algorithms include the brute force algorithm, dynamic programming methods, and various other approaches, all of which are exponential in time complexity. The TSP is NP-hard.

There are also many TSP algorithms that approximate the TSP tour in polynomial time. Some of these include the constructive heuristics like nearest neighbor, k-opt heuristic, and even artificial intelligence methods like ant colony optimization.

WSPs have been used to improve related problems. For example, WSPs were used to quickly find minimum spanning trees [2]. WSPs were used to build a graph in which Kruskal's algorithm was used to approximate the minimum spanning tree. Little work has been done for applying WSPs to TSP.

## 3 Brute Force with WSP Pruning

The brute force algorithm examines all (n-1)! permutations and takes the shortest permutation as the solution tour. This algorithm runs in exponential time, so it quickly becomes impractical in larger problems.

### 3.1. Algorithm

We present a quick and easy way to use WSPs to reduce the size of the permutation search space. At each step of the permutation branching, we prune away point choices that are well-separated from the set of the current point. The pseudocode is shown in Algorithm 1.

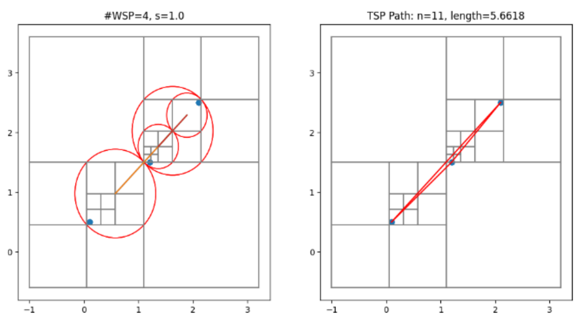

Figure 1: BFP run on data set with three clusters.

```
/* Build well-separated dictionary */
ws <- dict{point -> point set}
for wsp in wspd:
  for point pair (pA,pB) in wsp:
        ws[pA].add(pB)
        ws[pB].add(pA)

/* Spawn Permutation branching instances
   starting from each point */
permutations <- []
for point in points:
  rem <- TSP points
  initPerm <- [point]
  perms <- buildPerm(initPerm, rem)
  permutations += perms

/* Build Permutations by branching to points
   that are not well-separated */
func buildPerm(perm, rem):
  perms = []
  if len(rem) == 0:
    perms += perm
  last <- last item from perm
  while len(rem) > 0:
    for next in rem:
      if ws[last] not contains next
      nextPerm <- perm.add(next)
      nextRem <- rem.remove(next)
        perms += buildPerm(nextPerm, nextRem)
  return perms
```

Algorithm 1: Brute force with WSP Pruning.

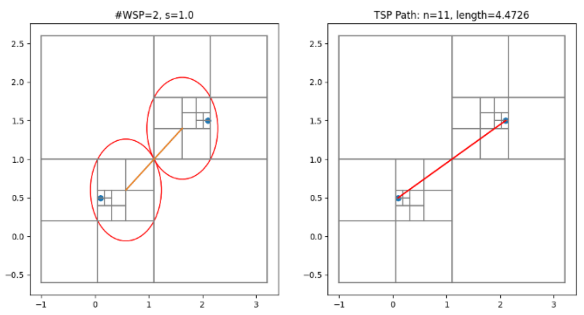

Figure 2: BFP run on data set with two clusters.

## 3.2. Performance

By pruning away choices, we separate the larger problem into many smaller problems. In Table 1, we show the performance improvements on different data sets. We see more time savings for point sets with well-separated clusters. Each cluster is treated like a separate problem and cheaply connected to the points not in the cluster.

The data set with three clusters (Figure 1) only checks 5,472 permutations compared to the one with two clusters (Figure 2), with 17,280 permutations. Three WSPs were found in the data set with three clusters. This breaks the 11-point problem into a three point and two four-point subproblems. A single WSP was found in the data set with two clusters. This breaks the 11-point problem into a five point and a six-point subproblem.

Even with these reductions in permutation space, the brute force algorithm is still exponential worst-case. Next, we investigate a polynomial time algorithm.

| Data set | Maximum permutations | Permutations with WSP pruning |
|---|---|---|
| 3 Clusters N=11 | 3,628,800 | 5,472 |
| 2 Clusters N=11 | 3,628,800 | 17,280 |
| Uniform N=11 | 3,628,800 | 1,406,160 |

Table 1: Permutations checked for the Brute Force Algorithm with and without WSP Pruning. The data sets can be found in the GitHub repository.

## 4 WSP Subproblem Algorithm

In practice, the optimal tour is not always necessary. We introduce a new polynomial-time approximation algorithm that combines the brute force algorithm and nearest neighbor algorithm with subproblems formed by well-separated pairs.

### 4.1. Motivation and Overview

A problem with the regular nearest neighbor method is that by greedily taking the nearest neighbor as the next point of the tour, it often leaves points in corners of the space. It must backtrack after leaving the neighborhood sometimes causing long jumps near the end of the tour. To prevent this behavior, we take a top-down approach breaking the problem into multiple recursive subproblems. We force the tour to visit every point in a neighborhood before visiting another. Each neighborhood becomes a subproblem. This process happens within subproblems until it cannot be broken down further. Then we connect subproblems.

### 4.2. Finding Subproblems

We use the well-separated pair decomposition of the point set to break the space into subproblems. Starting from the largest WSPs, we build pairs of point sets that we split the data by. All the points within the primary WSP set belong in the first set. The second set is found by taking all points from the secondary WSP set and all the secondary WSP sets from outgoing WSPs from within the primary WSP set. Figure 3 illustrates this more clearly.

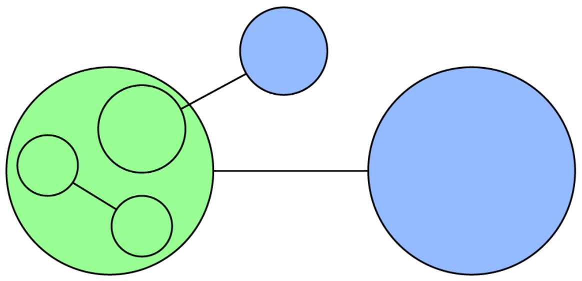

Figure 3: The pair of circles connected by a line represent a WSP. Points from green region are added to first set of split pair while those from the blue region.

Next, the split pairs are sorted such that those splitting the most number of points most evenly rank first. We apply each split pair to the original point set in this order. Points belonging to either of the sets from the split pair are grouped into sublists. Each split pair is applied top-down into each sublist recursively. When all split pairs are applied, we get a list of grouped multi-tiered sublists that we will use as our subproblems. Pseudocode for both finding and applying the split pairs is in Algorithm 2.

```
/* Build split set pairs from WSPD */
splitSetPairs <- [(point set, point set)]
for block in WSP-quadtree:
  curSet <- block.getAllPoints()
  wsSet <- []
  for wspBlock in block:
    wsSet.add(wspBlock.getAllPoints())
  splitSetPairs.add(curSet, wsSet)

/* Build subproblems with splits */
func applySplitPair(list, pair):
  subprob1 = []
  subprob2 = []
  for item in list:
    if item is of type list:
      applySplitPair(item, pair)
    else if item is of type point:
      if item in subprob1:
        subprob1.add(item)
        list.remove(item)
      else if item in subprob2:
        subprob2.add(item)
        list.remove(item)
  list.add(subprob1, subprob2)

subprobs <- [set of unordered data points]
for pair in splitSetPairs:
  applySplitPair(subprobs, pair)
```

Algorithm 2: Pseudocode for building subproblems in the WSP Subproblem Algorithm.

### 4.3. Connecting and Solving Subproblems

Connecting subproblems is the same as solving subproblems. We connect subproblems in a top-down fashion. Each subproblem contains a set of points. At each level, we search for a high-level path connecting the subproblems. We run the brute force algorithm when the subproblem size is small and the nearest neighbor algorithm when the subproblem size is large. A non-brute force optimal algorithm can be used here, but we chose the brute force algorithm for its simplicity and our time constraints. In either algorithm, we treat each subproblem like a single point. Instead of taking the distance between points, we take the minimum projection between subproblems. The minimum projection is the closest pair of points between two sets [2]. For each subproblem in the high-level path, we have an entry and exit point found by the minimum projections to its preceding and succeeding subproblem. We use the entry and exit points to recursively solve each subproblem, returning a path of the TSP tour. Figure 4 illustrates connection between subproblems.

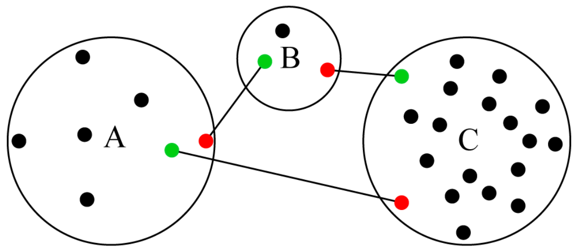

Figure 4: The circles represent subproblems at one level. Each subproblem contains further subproblems. Minimum projections are drawn as lines connecting the subproblems at the red (exit) and green (entry) points.

The pseudocode is shown in Algorithm 3. There are many edge cases not shown here. We invite the reader to look at our full implementation found in the GitHub repository.

```
/* Connect and solve subproblems */
func connSubprobs(entry,subprobs,exit):
  entryItem <- subprob that contains entry
  exitItem <- subprob that contains exit
  rem <- subprobs
  rem.remove(entryItem)

  subprobPath <- {
    starting from entryItem …
    ending at exitItem …
    if len(subprobs) < threshold:
      brute force w/ min projection
    else:
      nearest neighbor w/ min projection
  }

  path = []
  for (entry,subprob,exit) in subprobPath:
    path += connSubprobs(entry,subprob,exit)

path <-
  connSubprobs(any point, subprobs, any point)
```

Algorithm 3: Pseudocode for connecting subproblems in the WSP Subproblem Algorithm.

## 4.4. Performance

In Table 2, we show the results from our implementation of the WSP Subproblem algorithm. The solutions range from 12.5% to 38.1% longer than the optimal tour length on the datasets we tried. Figures 5-6 show the resulting tour for some of the datasets.

| Data set | Optimal Tour Length | WSP Subproblem Alg. Tour Length |
|---|---|---|
| ATT48 | 33,523 | 37,718 **(+12.5%)** |
| XQF131 | 564 | 769 **(+36.3%)** |
| XQG237 | 1019 | 1345 **(+32.0%)** |
| PMA343 | 1368 | 1834 **(+38.1%)** |
| PR1002 | 259,045 | 328,148 **(+26.7%)** |

Table 2: Tour lengths from the WSP Subproblem Algorithm. Percentage indicates length over optimal tour. The data sets can be found in the GitHub repository.

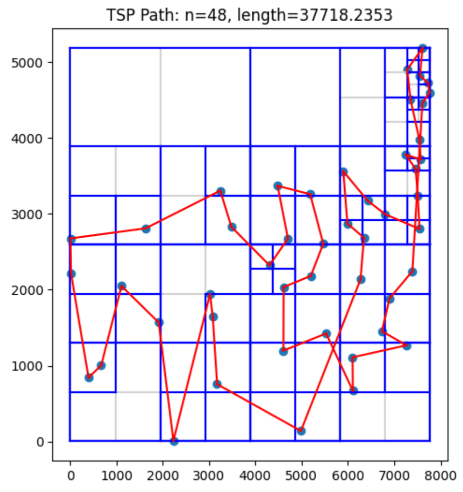

Figure 5: Best tour for ATT48 found by WSP Subproblem algorithm using PK PMR tree.

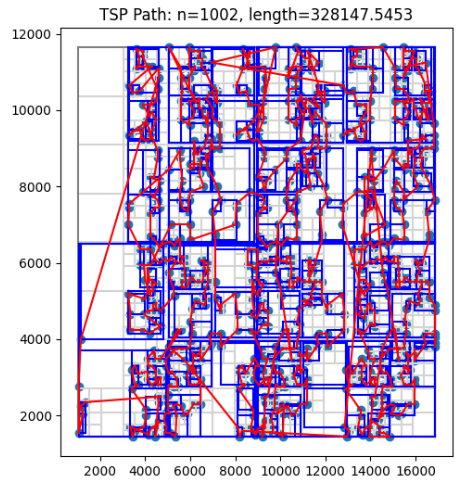

Figure 6: Best tour for PMA1002 found by WSP Subproblem algorithm using PK PMR tree with block shrinking.

### 4.5. Hardness

The results from Table 2 are found after tuning hyperparameters such as separation factor and split pair ranking metric. The quality of solutions from the WSP Subproblem algorithm is sporadic. It is dataset dependent and sensitive to hyperparameters. In this section, we try to quantify which characteristics produce good or bad solutions.

Separation factor plays an important role in finding the WSPD. Small separation factors result in WSPs with larger radiuses that contain more smaller WSPs within. Large separation factors result in WSPs with small radiuses that are farther apart and have less WSPs within.

The split pair ranking metric is a weighted score between average set count and difference between set counts. More weight towards the difference results in larger and more hierarchical subproblem groupings.

There is no single configuration of hyperparameters that produce the best results for all datasets. As of now, the best configurations vary from problem to problem and are found with trial and error.

### 4.6. Alternative Non-WSP Tree Method

Using the WSPD to build our subproblems is motivated by the idea that the WSPD divides the problem into subproblems. A variation of this algorithm can be performed without WSPs. If we just build our subproblems by traversing the quadtree and taking the points from each node, we achieve similar results for some cases. The solutions found by this are consistent, but rarely beat the WSP variation with tuned hyperparameters.

## 5 Quadtrees and WSPD

Building the well-separated pair decomposition often takes longer than some polynomial-time TSP algorithms. We perform a study on four types of quadtrees (Point, PR, PMR quadtrees, and their PK quadtree variants) to find the one that best fits the WSP-TSP setting. We evaluate each tree based on the resulting TSP tours and WSPDs.

### 5.1. Point Quadtree

The Point quadtree is the simplest to implement. Every node in the tree contains a single data point also marking where the block splits. Because point quadtrees split at the points, the shape of the blocks depends on the data and the order that data is inserted into the tree. As a result, we may get oddly shaped long rectangular blocks. The diameter of a WSP is determined by the diagonal of the quadtree block. This makes long rectangular blocks bad for WSPDs as it prevents some pairs of blocks from being considered WSPs.

### 5.2. PR Quadtree

The PR quadtree had square blocks, favorable for good WSPD. Nodes split when the number of points it contains exceeds a bucketing threshold. It splits into four equal quadrants until the points are separated under the threshold in the resulting child nodes. As a result, the PR quadtree has depth limitations. When points are clustered closely, the tree can become very deep.

### 5.3. PMR Quadtree

The PMR quadtree is based off the PM quadtree. Like the PR quadtree, a PMR node splits into four equal quadrants when it exceeds the splitting threshold. However, the PMR node splits once and only once. This splitting behavior limits the depth of the tree to the number of points. As a result, the PMR quadtree has adaptive bucketing. When points are clustered together, they can be grouped into a single bucket regardless of the splitting threshold. When points are far apart, they are separated when the node splits like in the PR quadtree.

The PMR quadtree has the favorable features, but none of the limitations from the Point and PR quadtrees.

## 5.4. PK Tree

Like how the PR quadtree splits when more than a k points in a node, the PK tree groups when less than k points in a node [3]. The PK tree is built in two phases: partitioning and aggregation. The partitioning phase is the same as building any other quadtree. It can use any type of quadtree. For our project, we used both the PR and PMR quadtrees. The aggregation phase bottom-up removes nodes with children that have less than a total of k points and reassigns the children to its parent. The PK tree removes empty nodes and combines low data count nodes.

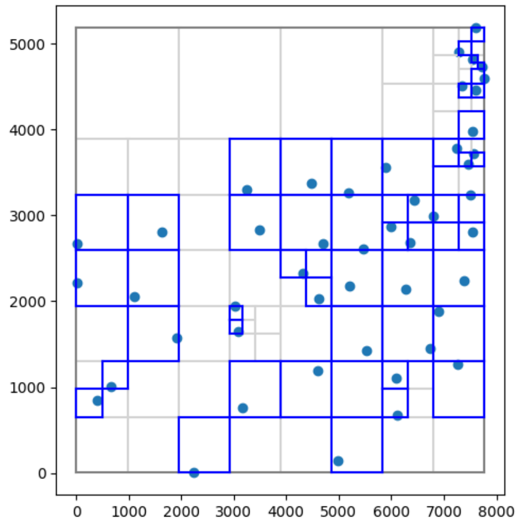

Figure 7: Visualization of the PK tree on ATT48. Blue blocks are the nodes aggregated into the PK tree. The gray blocks are the divisions created by the PR quadtree during the partitioning phase.

## 5.5. Block Shrinking

Because the radius of a WSP is determined by the diagonal length of the tree block, it is in our best interest to keep it as small as possible to ensure all WSPs that should meet the WSP condition are found. An easy way to reduce the block sizes is to shrink the block boundaries to fit the range of the points. This can be found in linear time.

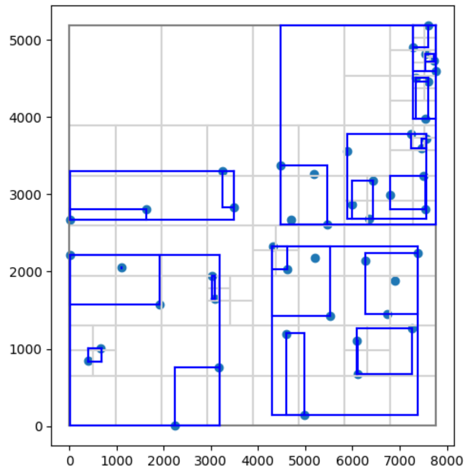

Figure 8: Visualization block shrinking on a PK tree of ATT48.

## 5.6. Performance

Runtimes for building the WSPD on multiple datasets using the three quadtree types can be found in Table 3. Although point quadtrees generally run faster, they form bad WSPDs for TSP. The PR quadtree produces decent WSPDs but fail when points are clustered too closely together. The PMR quadtree performs slightly faster, with similar quality of WSPD, and no worst-case depth limitations.

| Data set | Point | PR (bucket=1) | PMR |
|---|---|---|---|
| ATT48 | 0.9 sec | 1.37 sec | 1.34 sec |
| XQF131 | 2.4-3.5 sec | 5.2 sec | 4.8 sec |
| XQG237 | 5.2-8.8 sec | 9.7 sec | 9 sec |
| XQL662 | 24-33 sec | 38.0 sec | 36.0 sec |
| Worst-Case222 | 2.39 sec | Python Recursion Limit Reached | 1.26 sec |

Table 3: Runtimes for generating WSPD using various types of quadtrees. Worst-Case222 is a custom dataset that contains multiple clusters points “microscopically” close together.

## 6 Conclusion

We demonstrated an easy application of WSPs to reduce the search permutation space of the brute force algorithm. With these computational savings, it becomes possible to run the brute force algorithm on some datasets that were previously too long. We also introduced a new polynomial time algorithm that uses the WSPD to break large TSP problems into smaller subproblems. It connects subproblems top-down with existing algorithms. Currently, our this algorithm performs poorly on some datasets and hyperparameters. WSPs may not be a great structure used to constructively build subproblems. When more consistency is needed, the variation discussed in 4.6 should be used. However, WSPs were incredibly useful for pruning search space.

We performed a study on four types of quadtrees and determined that the PMR quadtree was best suited for finding WSPDs for the TSP.

Python implementations for all algorithms and trees in this paper can by found in the GitHub repository.

https://github.com/JerryGQian/WSP-TSP/

With limited time, there are some areas we did not explore. Like the brute force improvement, WSPs may be useful for improving the performance of other existing algorithms.

## Acknowledgements

The author would like to thank Jagan Sankaranarayanan and Prof. Hanan Samet for their guidance throughout the project.

## References

[1] D. Mount, *CMSC 754: Lecture 16 Well-Separated Pair Decompositions*, 2020. [Online].

[2] C. Li, *Euclidean Minimum Spanning Trees Based on Well Separated Pair Decompositions*, 22-May-2014. [Online].

[3] H. Samet, *DECOUPLING: A FREE (?) SPATIAL LUNCH*, 2002. [Online].