Metric Distances of Permutations

Ananya Das
Department of Computer Science
Columbia University
ad726@columbia.edu
and
Mark Roberts
Department of Computer Science
Colorado State University
mroberts@cs.colostate.edu

July 15, 2004

Abstract

Modeling the cost of algorithms on optimization problems has recently been a topic of considerable interest in the field of Artificial Intelligence. One key factor in this modeling is determining how "close" one solution is to another, or in other words, determining the distance between two solutions. When the solutions are just permuations of numbers, we can determine the distance between two solutions by calculating the distance between the two corresponding permutations. We explore various metric distances between two solutions: precedence distance, adjacency distance, positional distance, and swapping distance. We compute these distances and compare the values. We further explore swapping distance by studying the paths taken to transform one permutation to another. We evaluate these paths to draw conclusions about the distances between two solutions.

1 Introduction

The job-shop scheduling problem (JSP) is one of the most difficult NP-complete problems known. For this reason, it is widely studied in the field of Artificial Intelligence. A solution to this problem is a schedule which can be represented as an ordering (or permutation) of the jobs. Therefore, a solution to the problem of scheduling n jobs can be represented as a permutation of the numbers from 0 to n-1. We may be able to derive conclusions about the solutions of an instance of JSP by comparing these solutions. We can determine how "close" one solution is to another by calculating the metric distance between the permutations of the solutions. In this paper, we study the *precedence*, *adjacency*, *positional* and *swapping* distances of permutations.

Finding the swapping-based metric distance between two permuations has an interesting side effect. We start with an initial permutation. At each step we swap two of the jobs.

We continue until we arrive at our final permutation. We call the series of permutations we obtain through this process the "path" between swapping. It is interesting to study each step of the path obtained by swapping from one permutation to another. In this paper, we record the paths obtained between solutions and evaluate each step of the paths in order to get a better understanding about the solutions.

2 Precedence Distance

Suppose we would like to find the distance between two permutations π and $\pi\prime$, where π and $\pi\prime$ consist of n jobs. The *precedence metric* counts the number of times a job j is preceded by a job i in both π and $\pi\prime$. The precedence distance is this quantity subtracted from $n(n-1)/2$. The quantity $n(n-1)/2$ represents the number of pairwise swaps that can be performed on a permutation [2].

The *precedence metric* distance is found by first creating the precedence matrices for both π and $\pi\prime$. Each matrix is of size n x n. If job i precedes job j in π, then a 1 is placed in entry (i, j) of $\pi_{precMatrix}$. Otherwise, a 0 is placed in entry (i, j). The same rule applies for the precedence matrix of $\pi\prime$. Once the two precedence matrices are created, the *precedence metric* distance can be found by comparing the corresponding entries of the two matrices. If entry (i, j) of $\pi_{precMatrix}$ is set to 1, and entry (i, j) of $\pi\prime_{precMatrix}$ is also set to 1, then job i precedes job j in both π and $\pi\prime$. The number of times this event occurs is counted and subtracted from $n(n-1)/$. The final difference is the *precedence metric* distance [3].

```
getPrecedenceMatrix(π, n)
//Returns the precedence matrix of permutation π.  n is the number of jobs in π.
for ( i ← 0 to n)
 for ( j ← 0 to n)
  precedenceMatrix[i][j] ← 0
for ( i ← 0 to n)
 for ( j ← i + 1 to n)
  precedenceMatrix[π_i][π_j] ← 1
return precedenceMatrix
```

Figure 1: Pseudocode for the getPrecedenceMatrix

3 Adjacency Distance

The *adjacency metric* counts the number of times a pair i, j of jobs is adjacent in both π and $\pi\prime$. Suppose we let n_{adj} equal this quantity. The adjacency distance is then $n - n_{adj} - 1$ [2].

PrecedenceDistance(π, $\pi\prime$, n)
//Returns the precedence based distance between π and $\pi\prime$. n is the number of jobs in both π and $\pi\prime$.

```
π_precMatrix = getPrecedenceMatrix(π)
π′_precMatrix = getPrecedenceMatrix(π′)
  n_prec ← 0
  for ( i ← 0 to n)
   for ( j ← 0 to n)
    if (i ≠ j)
     if π_precMatrix[i][j] = 1 and π′_precMatrix[i][j] = 1
       n_prec ← n_prec + 1
d ← n(n − 1)/2 − n_prec
return d
```

Figure 2: Pseudocode for computing *Precedence Metric* Distance

The *adjacency metric* distance is found by first creating the adjacency matrices for both π and $\pi\prime$. Each matrix is again of size n x n. If job i is adjacent to job j in π, then a 1 is placed in entry (i, j) of $\pi_{adjMatrix}$. Otherwise, a 0 is placed in entry (i, j). The same rule applies for the adjacency matrix of $\pi\prime$. Once the two adjacency matrices are created, the *adjacency metric* distance can be found by comparing the corresponding entries of the two matrices. If entry (i, j) of $\pi_{adjMatrix}$ is set to 1, and entry (i, j) of $\pi\prime_{adjMatrix}$ is also set to 1, then job i is adjacent to job j in both π and $\pi\prime$. We let $n_a dj$ equal the number of times this event occurs. The *adjacency metric* is then found by computing the value of $n - n_{adj} - 1$ [3].

getAdjacencyMatrix(π, n)
//Returns the adjacency matrix of permutation π. n is the number of jobs in π.

```
for ( i ← 0 to n)
 for ( j ← 0 to n)
  adjacencyMatrix[i][j] ← 0
for ( i ← 0 to n)
  adjacencyMatrix[π_i][π_{i+1}] ← 1
return adjacencyMatrix
```

Figure 3: Pseudocode for getAdjacencyMatrix

AdjacencyDistance(π, $\pi\prime$, n)

//Returns the adjacency based distance between π and $\pi\prime$. n is the number of jobs in both π and $\pi\prime$.

$\pi_{adjMatrix}$ =*getAdjacencyMatrix*(π)

$\pi\prime_{adjMatrix}$ =*getAdjacencyMatrix*($\pi\prime$)

$n_{adj} \leftarrow 0$

for ($i \leftarrow 0$ to n)

 for ($j \leftarrow 0$ to n)

  if ($i \neq j$)

   if $\pi_{adjMatrix}[i][j] = 1$ and $\pi\prime_{adjMatrix}[i][j] = 1$

    $n_{adj} \leftarrow n_{adj} + 1$

$d \leftarrow n - n_{adj} - 1$

return d

Figure 4: Pseudocode for computing *Adjacency Metric* Distance

4 Positional Distance

The *positional metric* compares the actual positions in the sequence of job j in both π and $\pi\prime$. Given the permutation π, its inverse permutation σ gives the position of job j. For example, suppose job j is in the i^{th} position of π. Then $\sigma_j = i$. The positional distance is then found by computing

$$\Sigma_{j=1}^{n}|\sigma_j - \sigma\prime_j|$$

[1]

The *positional metric* distance is found by first creating the inverses σ and $\sigma\prime$ of both π and $\pi\prime$ as described above. The inverses are stored in arrays. The absolute values of the differences of the entries in the arrays are then summed up to find the positional distance.

getInverseArray(π, n)

//Returns the inverse of permutation π stored in an array. n is the number of jobs in π.

for ($i \leftarrow 0$ to n)

 $j = \pi_i$

 $\sigma_j \leftarrow i$

return σ

Figure 5: Pseudocode for getInverseArray

```
PositionalDistance(π, π′, n)
//Returns the positional based distance between π and π′.  n is the number of jobs
in both π and π′.
1.  σ =getInverseArray(π)
2.  σ′ =getInverseArray(π′)
3.  d ← 0
4.  for ( i ← 0 to n)
5.   d ← d + abs(σ_i − σ′_i)
6.  return d
```

Figure 6: Pseudocode for computing *Positional Metric* Distance

5 Swapping Distance

The *swapping metric* counts the minimum number of transpositions of two adjacent characters necessary to transform one permutation into another. Given a permutation π, a swap at position i produces the permutation $\pi_1, \pi_2, \ldots \pi_i, \pi_{i-1}, \pi_{i+1}, \pi_{i+2} \ldots \pi_n$.

The swapping algorithm works as follows: In order to transform permutation π into permuation $\pi\prime$ using swaps we must first map $\pi\prime$ into a standard permutation. We use the permutation $\sigma\prime = 0, 1, 2, \ldots n$ where n is the total number of jobs in π. Once we have mapped $\pi\prime$ into $\sigma\prime$, we use this mapping to map π into a new permutation σ. Now, we perform swaps to transform σ into $\sigma\prime$. The minimum number of swaps necessary to transform σ into $\sigma\prime$ is the swapping distance for this transformation [1]. By mapping the final permuation $\pi\prime$ to a standard permutation, we allow ourselves to have a consistent method of counting the number of swaps of any transformation.

Each swap along the transformation produces a different permuation. We define the set of consecutive permuations obtained from swapping to be the swapping "path" and each permuation obtained to be a "step" along the path. The swapping distance between π and $\pi\prime$ can also be seen as the number of steps along the path between π and $\pi\prime$. To get a better understanding of the differences between π and $\pi\prime$ we evaluate each step along the path and compare the evaluations.

We define a "bump" to be the number of jobs that the scheduling algorithm is unable to schedule given a permutaition π. Our evaluation function counts up the total number of bumps in every step along a swapping transformation from π to $\pi\prime$.

6 Results and Conclusion

For the dataset "Day12gen_bestSolutions" we computed the precedence, adjacency, positional, and swapping distances for every pair of permutations. To compare the distances, we took an average of each type of distance and plotted the averages. We did the same

```
SwapDistance(π, π′, n)
//Returns the swapping based distance between π and π′.  n is the number of jobs
in both π and π′.
1.  σ′ = 0, 1, 2, ..., n
2.  τ(x) = mapping from π′ to σ′
3.  σ = τ(π)
4.  for ( j ← 0 to n)
5.   i_π = pos(j) in π
6.   i_σ = pos(j) in σ
8.   if i_π ≠ i_σ
9.   do{
10.    swap π_i and π_{i−1}
11.    i ← i − 1
12.    d ← d + 1
13.   }while( i_π ≠ i_σ)
12.  return d
```

Figure 7: Pseudocode for computing *Swapping Metric* Distance

for the datasets "Day15gen_bestSolutions" and "Day18gen_bestSolutions". Figures 8-10 display our results.

Using the evaluation function described above, we evaluated the swapping path of the transformation from the first permutation to the second permutation of the dataset "Day12". To get an understanding of the path, we plotted the evaluation vs. the step for the path. We did the same for the datasets "Day15" and "Day18". Figures 11-13 display our results.

metric	*avg.distance*
precedence	21339.571515151514
adjacency	319.9349494949495
positional	28890.176565656566
swap	18194.17777777778

Figure 8 - Day12gen

metric	*avg.distance*
precedence	20420.55898989899
adjacency	315.9068686868687
positional	27693.8602020202
swap	20252.54525252525

Figure 9 - Day15gen

metric	*avg.distance*
precedence	18997.381818181817
adjacency	294.96909090909094
positional	25668.222626262625
swap	20819.66606060606

Figure 10 - Day18gen

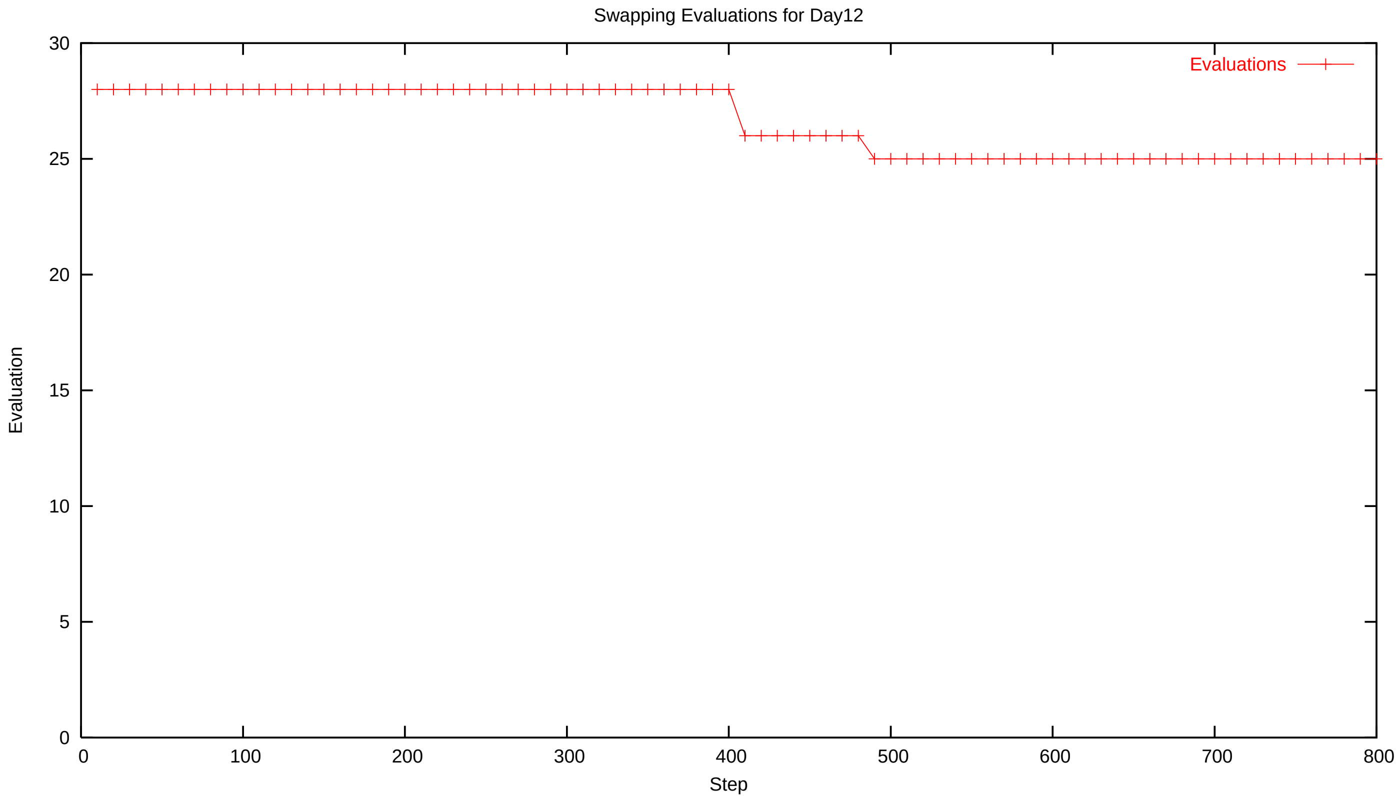

Figure 8: Evaluations vs. Step for Day12

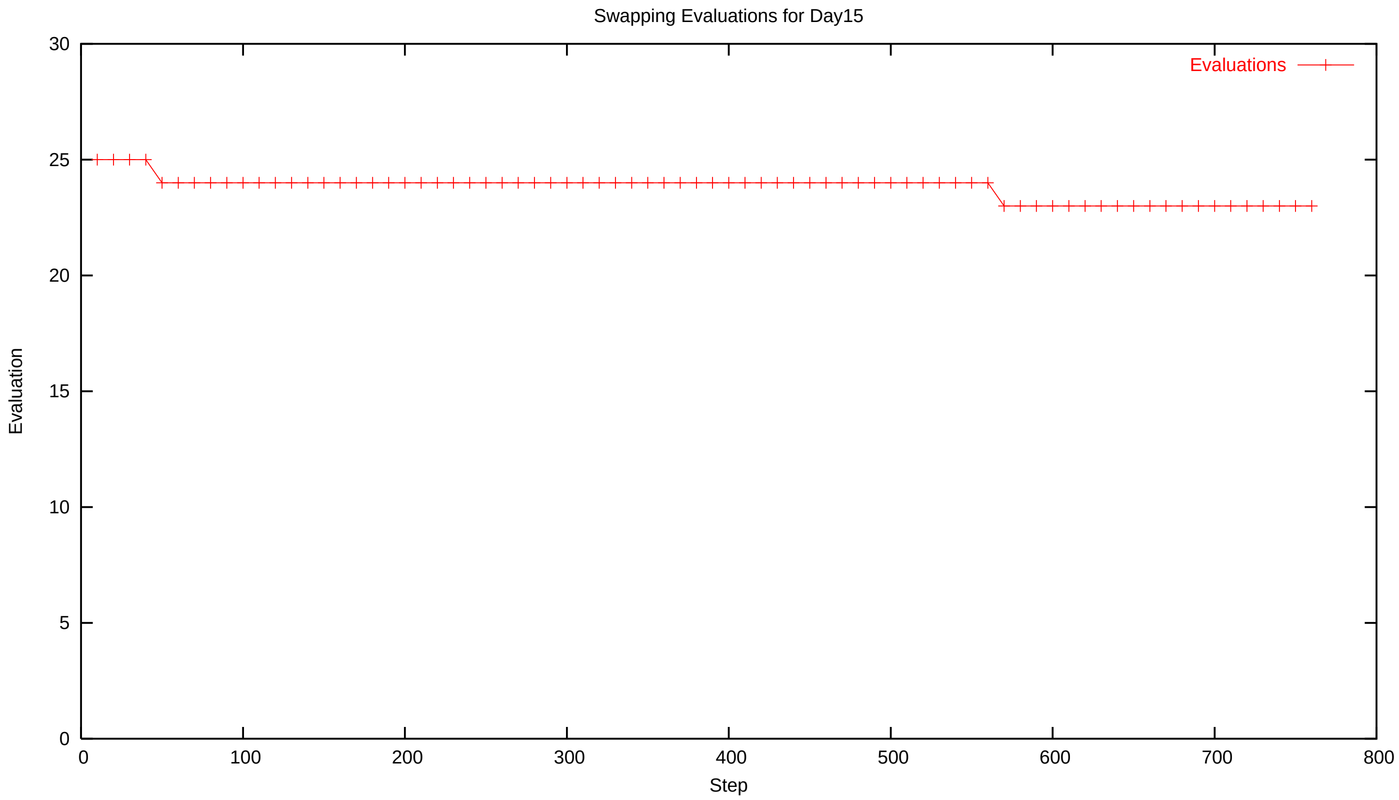

Figure 9: Evaluations vs. Step for Day15

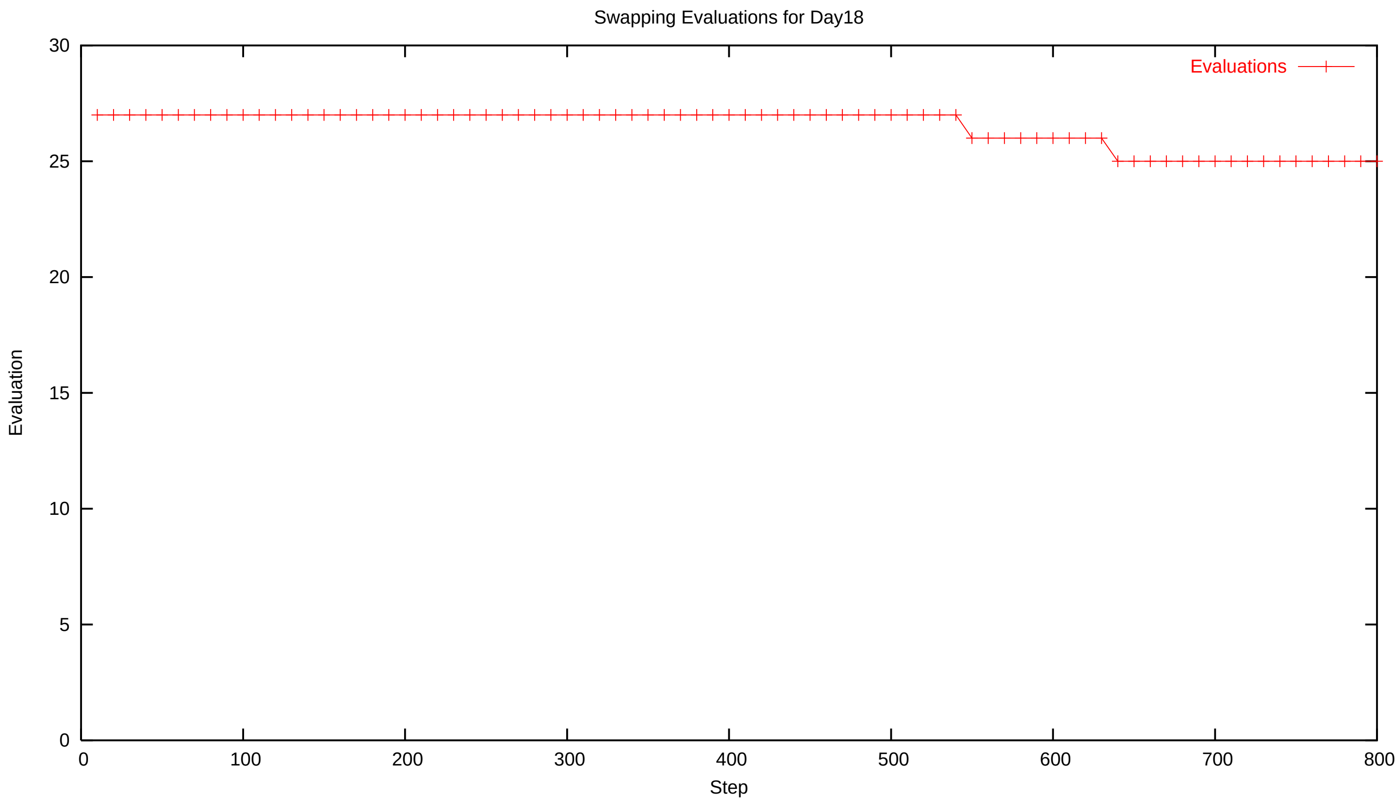

Figure 10: Evaluations vs. Step for Day18

Figures 8-10 show that the adjacency distance between two random permutations is much smaller than the precedence, positional, and swap distances. The figures also show that in general, the positional distance is the largest of the four distances.

Figures 11-13 show that as the step number increases (ie. as we approach $\pi\prime$), the evaluation decreases (less jobs are bumped). However, the evaluation seems to be relatively constant throughout the path.

The results of our data will be used as a comparative tool for JSP solutions. Analysis of the data will lead to insight for search topology.

7 References

1. Cormode, Graham. Sequence DIstance Embeddings. *Thesis submitted to University of Warwick*, pp 6-8, 49-50. (2003)

2. Reeves, Colin R. Landscapes, Operators and Heuristic Search, *Annals of Operational Research*, 86, pp 473-490. (1999)

3. Roberts, Mark. Colorado State University *Interview*