A Type of Variation of Hamilton Path Problem with Applications

Jitian Xiao
Edith Cowan University

Jun Wang
Wenzhou University, Zhejiang, China

A Type of Variation of *Hamilton Path* Problem with Applications

Jitian Xiao
School of Computer and Information Science, Edith Cowan University, Mount Lawley, WA 6050, Australia
E-mail: j.xiao@ecu.edu.au

Jun Wang
School of Computer Science and Computer Engineering, Wenzhou University, Zhejiang, China
Email: jsj_wj@wzu.edu.cn

Abstract

This paper describes a type of variation of the Hamilton Path problem that can be applied to a type of applications. Unlike the original Hamilton Path problem, the variation always has a solution. The problem of finding solutions to the variation of the Hamilton Path problem is NP-complete. A heuristic for finding solutions to the problem is developed and analyzed. The heuristic is then applied to a real application scenario in the area of spatial cluster scheduling in spatial join processing. Experiments have demonstrated that the proposed method generates better cluster sequence than existing algorithms.

Keywords: *Hamilton path, heuristic, scheduling, approximation.*

1. Introduction

The sequencing of vertices of a graph has been applied to many applications, such as cluster scheduling of spatial join operations in Geographic Information Systems (GIS) [7] and generation of shortest test sequences for software testing [5]. A distinct characteristic of these applications is the large amount of data they handle. A key issue in the performance of these applications is the use of fast algorithms.

In this paper, we formally define a type of variation of the Hamilton path (VHP) problem that can be applied to a type of applications. As the problem of finding solutions to the VHP problem is NP-complete, we develop a heuristic for finding an approximate solution to the VHP problem. And, a real application case study of the VHP problem is presented, with some initial experimental results against existing algorithms.

The rest of the paper is organized as follows: Section 2 defines the VHP problem. Some theoretical results of the VHP problems are also included in this section. The heuristic is presented and analyzed in Section 3. Section 4 describes an application scenario. Section 5 concludes the paper.

2. A variation of Hamilton path problem

A *Hamilton path* (HP) of a graph is a path between two vertices of the graph that visits each vertex exactly once. For a weighted graph G, an HP of G is a path between two vertices of G that visits each vertex exactly once, and the total weight of edges along the path is maximal (or minimal, respectively) among all such paths. The corresponding HP is sometimes called a *longest* (or *shortest*, respectively) *HP* of G. The problem of finding a (*longest* or *shortest*, respectively) Hamilton path in a (weighted) graph G is called the (*longest* or *shortest*, respectively) *HP problem* of G. A (longest, or shortest, respectively) HP of G is sometimes called a *solution* of the (longest, or shortest, respectively) HP problem. In this paper, we limit our discussion to the case of finding the longest HP problem (the shortest HP problem can be discussed similarly).

While many optimization applications, such as job scheduling, etc., can be modeled by a graph, these applications are hardly converted directly to a HP problem due to three reasons. Firstly, a graph may not contain a HP thus the HP problem may have no solution at all. Secondly, there is not a sufficient and necessary condition to determine whether or not there exists a solution to the HP problem of a (weighted) graph [6]. And, thirdly, even if a graph contains a PH solution, the algorithm of finding a PH solution is *NP*-complete. These properties of the PH problem have greatly limited its application.

We now describe a variation of the HP problem that applies to a type of applications and it always has a solution.

Definition 1. Given a weighted graph $G = (V, E, w)$ with $V = \{v_1, v_2, ..., v_n\}$, a (longest) *virtual (Hamilton)* *path* of G is defined as a sequence ($v_{i_1}, v_{i_2}, ..., v_{i_n}$) such that for all $v_{i_j}, v_{i_k} \in$ V, $v_{i_j} \neq v_{i_k}$ if $j \neq k$, and $\sum_{l=1}^{n-1} w_v(v_{i_l}, v_{i_{l+1}})$ reaches the maximum among all permutations of *V,* where w_v is defined as

$$w_v(v_i, v_j) = \begin{cases} w(v_i, v_j) & \textit{If there is an edge between } v_i \textit{ and } v_j \\ 0 & \textit{Otherwise} \end{cases} \quad (1)$$

w_v is called *virtual weight* function between vertices of G. If $(v_i, v_j) \notin E$ *(i.e.,* $w_v(v_i, v_j)=0$), we conceptually say that there is a *virtual edge* between v_i and v_j.

In other words, a (longest) virtual path of G is a permutation of vertices in *G* such that the sum of the weights of edges, if existed, between adjacent vertices along the virtual path reaches the maximum. The problem of finding a (longest) virtual path from G is called (longest) *virtual path (VP) problem* of G. The (longest) virtual path of G, i.e., the sequence ($v_{i_1}, v_{i_2}, ..., v_{i_n}$), is called a *solution to the VP problem*, and $\sum_{l=1}^{n-1} w_v(v_{i_l}, v_{i_{l+1}})$ is called the *(total) weight* of the VP solution.

If a graph is unweighted, we can convert it to a weighted graph by simply defining a weight function *w* as $w(v_i, v_j)=1$if there is an edge between vertices v_i and v_j, $1 \leq i, j \leq n$. Such a weighted graph is sometimes called *trivial* weighted graph.

The following properties hold for the VP problem (the proof is omitted here due to space limitation):

Property 1: For any graph G, the VP problem of G always has a solution.

Property 2: If a graph G is a trivial weighted graph and it has a HP, then a sequence of vertices in G is a HP solution if and only if it is a VP solution.

Property 3: The problem of finding a VP solution from a weighted graph is *NP*-complete.

Property 4: Let G be a graph. If each component G_i of G gets a VP solution $v_{i_1}, v_{i_2}, ..., v_{i_{m_i}}$ (i = 1, 2, …, p), then the sequence $v_{1_1}, v_{1_2}, ..., v_{1_{m_1}}, ..., v_{p_1}, v_{p_2}, ..., v_{p_{m_p}}$ is a VP solution of G.

Property 2 suggests that, for a trivial graph, the VP problem of G is a *generalization* of the HP problem in the sense that if G has a HP solution, then the HP solution is also a VP solution. However, the property does not hold for a weighted graph. This can be illustrated by the graph in Figure 1. While the sequence (v_1, v_4, v_3, v_5, v_2) is a HP solution of the graph (with a total weight of 4), it is not a VP solution of the graph. Instead, (v_1, v_4, v_5, v_2, v_3) is a VP solution of the graph, with a total weight of 7, and there is one virtual edge (v_2, v_3) in the VP solution.

Property 3 suggests that, although there exists a VP solution for each (weighted) graph, it is impossible to find a VP solution in polynomial time. From Property 4, the task of finding a VP solution from G can be reduced to the case where *G* is a connected graph.

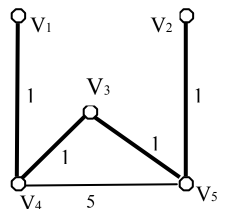


Figure 1. A HP solution but not a VP solution

3. A heuristic for finding a VP solution in a weighted graph

From Definition 1, a simplest algorithm to find a VP solution from a graph is to check all permutations of *V* to see which one makes the max{ $\sum_{l=1}^{n-1} w_v(v_{i_l}, v_{i_{l+1}})$ }. The complexity of such a method clearly has factorial order and is certainly not practical.

We now describe a heuristic to find a VP solution from a weighted graph. Our goal in this exercise is to build an efficient algorithm that not only generates a good quality *approximation* of a VP (AVP) solution (i.e., the total weight of the AVP solution is close to that of the VP solution), but also has a lower computation complexity.

For simplicity, we limit our discussion to connected graphs. From Property 4, the heuristic/algorithm and the related discussion can be easily extended to the case of unconnected graphs.

The following terminologies [6] are frequently used in the rest of this paper.

A *simple path* of a graph is a path in which all vertices are distinct. A *path graph* with *n* vertices is a graph in which all vertices can be listed as a sequence $v_1, v_2, v_3, ... v_{n-1}, v_n$ such that $(v_1, v_2), (v_2, v_3), ... (v_{n-1}, v_n)$ are the only edges of *E*. A *match* of a graph is a set of edges, in which any two of them are not incident to the same vertex. A match is *maximal* if any edge in the graph that is not in the match has at least one of its endpoints matched, and the sum of the edge weights of the match is maximal among all matches of the graph. A *weighted matching* (WM) problem is, for a given weighted graph *G*, to find a match of *G* such that the sum of the edge weights of the match is maximal. The WM problem was first solved by Edmonds [4] and the complexity of his algorithm is $O(n^2m)$, where *n* and *m* denote the number of vertices and edges, respectively, of the graph. Since then Edmonds' algorithm has been studied by a number of researchers. The fastest implementation of the Edmonds's algorithm is due to Cook and Role [3] with a time complexity of $O(nm \log n)$. For any graph, these algorithms output a *maximal match* of the graph.

The basic idea of our heuristic can be expressed as two phases: (1) to divide the graph into sets of disjoint path graphs such that the sum of the weights of the longest paths in the path graphs reaches the maximum; and (2) to continually link these paths using maximal match among the endpoints of the paths until no more match can be found. At this point, the final VP solution is formed by linking all longest paths of the resultant path graphs.

The first phase of the heuristic is a recursive algorithm component containing three main steps: In the first step, a maximal match M of G is produced using Cook's *maximum matching* algorithm [3]. Each edge, together with its connected vertices, in M is taken as an initial path graph. That is, G is conceptually split into a set of path graphs, each consisting of a pair of matched vertices and the edge connecting them. If more than one isolated vertices was left after matching, they are *conceptually matched* by randomly putting pairs of such (isolated) vertices together, or, matched using virtual edges. As such, there might be remained at most one unmatched vertex, which forms a special path graph (i.e., one without any edge).

In the second step, the graph G is coarsened by collapsing the matching vertices (or, endpoints of the longest paths in path graphs). At this step, each pair of matching vertices (or, endpoints of the path in individual path graph) are combined to form a single vertex of the next level coarser graph $G' = (V', E', w')$. Vertices in V' are all in form of either $v = \{v_i, v_j\}$, where $v_i, v_j \in V$ are (virtually) matched in M (i.e., $(v_i, v_j) \in M$), or in form of $v = \{v_i\}$, where v_i is a unmatched vertex of M (note that there is at most one such form of vertex for each level of coarsened graph).

Intuitively, each vertex $v=\{v_i, v_j\} \in V'$ represents a path graph in G where v_i and v_j are the endpoints of its longest path. A vertex v of form $\{v_i, v_j\} \in V'$ is referred to as a *t-vertex*, and a vertex v of form $\{v_i\} \in V'$ is referred to an *s-vertex*. A *multinode* can be either a t-vertex or an s-vertex.

E' and w' are then defined such that the edge between any pair of multinodes v' and v'' corresponds to an edge in E whose two endpoints are in v' and v'', respectively, and whose weight is maximal among all edges connecting nodes in between v' and v'' (i.e., the endpoints of the path graph represented by $\{v_i, v_j\}$), if such an edge exists.

After G' is built, Cook's maximal matching algorithm is applied to G' again to produce a maximal match M'. By this point, the next level of coarser graph $G'' = (V'', E'', w'')$ can be built following the same procedure (as described above). The above match-and-collapse process continues until no further matching can be found.

In the third step, any *t-vertex* $\{v_x, v_y\}$ in the last coarser graph can be stretched to a path graph, with the two endpoints as v_x and v_y, respectively. For a graph consisting of multiple components, the AVP solution is produced by printing vertices in all path graphs (i.e., the vertices in each path graph are listed in an order in its longest path), one after another.

In summary, we conceptually take a pair of matching vertices and the edge between them as a path graph at the end of first round of match-and-collapse process (i.e., each with a path of length 1 or 0), then from the second round of match-and-collapse process on, the longest paths in these path graphs are concatenated pairwisely using edges of maximal weight between the endpoints of the paths. With the matching and collapsing process going on, paths are linked using the maximal matching on levels of coarser graphs until a set of disconnected path graphs is reached. At this stage, a sequence of vertices of the longest path for each path graph was output. Any order of these sequences can be taken as an AVP solution, because the produced path graphs are disjoint with each other, and each vertex of the original graph belongs to one and only one path graph.

The heuristic can be informally described using C-like pseudo-code as below:

```
**Algorithm** MaxMatchAVP(G)
**Input**: $G = (V, E, w)$;  // A weighted graph with
                              // $V = \{v_1, v_2, ..., v_n\}$.
**Output**: $V_{i_1}, V_{i_2}, ..., V_{i_n}$ ;  // An AVP solution of G, a
                              //permutation of vertices in V.
[1] Find a maximal match M of G using Cook's
    algorithm;      // see reference [3]
[2] for all unmatched vertices
[3]     randomly choose a pair of vertices, and put
        them in M  // virtual matching
[4]   until no more virtual match can be made
              // conceptually match isolated vertices
              // using virtual edges
[5] if no matching was found
[6]   {for each isolated multimode
[7]        output its vertices in the order in its
           longest path;
              //output the AVP solution of one
              // path graph (connected component);
[8]   return};
```

```
    [9] Coarsen $G$ by collapsing matching vertices of $M$
        to produce a coarser graph $G'$;
    [10] $MaxMatchAVP(G')$;
    [11] **return**;}
}
```

Suppose that a weighted graph have n nodes and m' edges. Line 1 needs $O(n \cdot m' \log n)$ running time (refer to [3]). Lines 2~4 would be executed no more than $n/2$ times and needs at most $O(n)$ time as it scans at most once for each unmatched vertex to virtually match to another unmatched vertex, if existed. Lines 6~8 would be executed once only (i.e., when no more matching can be found) and needs at most $O(n^2)$ time as it scans at most once for each vertex to find one of the endpoints in each path graph, and then use constant time to find each one linked vertex after that. Lines 9 has the complexity of $O(n^2)$ because, for each matched vertex, it needs no more than once scanning to combine to its matched one to form a multinode of the next level coarser graph. For ease of analysis, let $m=\max\{n, m'\}$. Then the total complexity of lines 1~9 is limited by $O(n \cdot m \cdot \log n)$. Line 10 completes the recursive execution of the algorithm. Based on this, it is not hard to proof that the complexity of the algorithm is $O(n^3 \log n)$ (detailed proof is omitted due to space limitation).

4. An application: cluster scheduling of spatial join operations in spatial databases

In spatial databases, spatial join queries usually access a large number of spatial objects [1, 2]. As spatial objects can be very large in size, they are usually stored in secondary storage, such as disks. To process a spatial join operation, the referred objects need to be fetched into the main memory for processing. The I/O cost can be very high for a single spatial join operation.

The I/O cost can be reduced by clustering joinable spatial objects and then scheduling the join-operations such that the number of times the same objects to be fetched into memory can be minimized. One of the key issues behind this approach is how to produce a good sequence, known as a *scheduler* [7], of clusters to guide the join-operation cluster scheduling. We now illustrate that, the problem of finding an effective scheduler can be converted to the problem of finding an AVP solution over an weighted graph, thus can be solved using the heuristic proposed in Section 3. When comparing with the algorithm proposed in [7], our new heuristic generates better cluster sequence than the existing algorithm in the sense that more fetching time used for fetching those overlapping objects of clusters can be saved.

4.1. Preliminary of spatial join processing

A spatial join operation may involve many (large) objects which cannot be all fetched into the main memory at the same time to complete the join. In such a case, the join is divided into many sub-join operations, each joining a subset of joinable objects. To further reduce the I/O cost, researchers also proposed two-phase join strategy, i.e., *clustering* candidate objects into clusters and then *joining* these clusters pairwisely. The clustering phase tries to cluster spatial objects such that they join as many other objects as possible within their cluster and join as few objects as possible across clusters [7, 8]. In the joining phase, these clusters are scheduled in a sequence such that a maximum number of overlapping objects between consecutive clusters can be reused in the memory when processing next cluster (i.e., the overlapping objects do not need to be fetched into memory again because they are already there). In this way, a significant reduction on disk access has been achieved and demonstrated through simulations [7].

4.2. Application problem definition

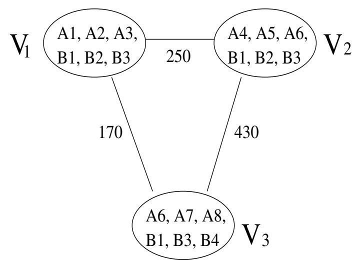
Suppose, for a given spatial join operation, the spatial objects involved have been clustered in the clustering phase. Let $\mathbb{V} = \{v_1, v_2, ..., v_k\}$ be the set of spatial objects referenced in the *candidate* set, and $V_1, V_2, \ldots, V_n$ the clusters of $\mathbb{V}$. For each i ($1 \le i \le n$), $V_i = \{v_{i_1}, v_{i_2}, \ldots, v_{i_m}\}$ ($m \ge 1$), $v_{i_j} \in \mathbb{V}$ ($1 \le j \le m$). That is, $\bigcup_{i=1}^{n} V_i = \mathbb{V}$ and $V_i \ne \Phi$ for each i ($1 \le i \le n$). For convenience, we define $size(V_i)$ as the sum of the sizes of objects in V_i, i.e., $size(V_i) = \sum_{v \in V_i} s(v)$ where $s(v)$ is the size of object v.

We introduce a weighted graph $G = (V, E, w)$, upon $\mathbb{V}$, called *cluster overlapping* (*CO*) graph, to represent the overlapping relationships between data clusters. The node set $V = \{V_1, V_2, ..., V_n\}$ is a set of clusters, and the edge set E is defined as: for each node pair V_i and V_j ($i \ne j$), there is an edge (V_i, V_j) if $w(V_i, V_j) = size(V_i \cap V_j) \ne 0$. Here $w(V_i, V_j)$ is the weight of the edge (V_i, V_j). As an example, let the spatial object set involved in a given spatial join operation be $\mathbb{V}$ = {A1, A2, A3, A4, A5, A6, A7, A8, B1, B2, B3, B4, B4}, the set of join operations be F = {(A1, B1), (A2, B1), (A3, B2), (A3, B3), (A4, B3), (A5, B1), (A6, B2), (A6, B4), (A7, B1), (A8, B3), (A8, B4)}, and its three clusters be V_1 ={(A1, B1), (A2, B1), (A3, B2), (A3, B3)}, V_2 ={

(A4, B3), (A5, B1), (A6, B2)} and V$_3$={ (A6, B4), (A7, B1), (A8, B3), (A8, B4)}. Based on the object sizes given in Figure 2 (a), Figure 2 (b) shows the CO graph corresponding to the above clusters.

When processing cluster V_{i+1}, objects in $V_i \cap V_{i+1}$ are already in memory just after processing V_i. There is no need to load these objects again. Thus, if the object clusters are joined in the sequence of $V_1, V_2, ..., V_n$ (i.e., no scheduling), then the total I/O cost is:

oid	obj size
A1	200
A2	80
A3	40
A4	30
A5	32
A6	260
A7	18
A8	60
B1	60
B2	80
B3	110
B4	76



(a) Object size (b) cluster overlapping graph

Figure 2. An example of CO graph

$$C_{I/O} = \sum_{i=1}^{n} size\,(V_i) - \sum_{i=1}^{n-1} size\,(V_i \cap V_j) \qquad (2)$$

Generally, for a schedule π which determines the processing sequence of $V_1, V_2, ..., V_n$ as $V_{\pi_1}, V_{\pi_2}, ..., V_{\pi_n}$, where $V_{\pi_i} \in V$ and $V_{\pi_i} \neq V_{\pi_j}$ for $i \neq j$, $1 \le i, j \le n$, the I/O cost for schedule π is

$$C^{\pi}_{I/O} = \sum_{i=1}^{n} size\,(V_{\pi_i}) - \sum_{i=1}^{n-1} size\,(V_{\pi_i} \cap V_{\pi_{i+1}}) \qquad (3)$$

When the clusters are given, $\sum_{i=1}^{n} size\,(V_{\pi_i})$ is a constant. The goal of cluster sequencing is to find a schedule π such that $\sum_{i=1}^{n-1} size\,(V_{\pi_i} \cap V_{\pi_{i+1}})$ is maximized, which is the case that $C^{\pi}_{I/O}$ is minimized.

4.3. Maximum overlapping order and AVP

The concept of *maximum overlapping (MO) order* was introduced in [7] to recognize better schedules. Given a CO graph $G=(V,E,w)$ with $V=\{V_1, V_2, ..., V_n\}$, an MO order among sets $V_1, V_2, ..., V_n$ is a sequence ($V_{i_1}, V_{i_2}, ..., V_{i_n}$ such that $\sum_{l=1}^{n-1} size\,(V_{i_l} \cap V_{i_{l+1}})$ reaches the maximum among all permutations of *V*. In other words, an MO order in a CO graph *G* is a permutation of nodes in *G* such that the total size of overlapping objects between adjacent nodes reaches the maximum. For example, (V_1, V_2, V_3) is an MO order in the CO graph in Figure 2 (a), and the total size of overlapping objects between adjacent nodes in the order is 680.

It is evident that the problem of finding an MO order from a CO graph is equivalent to that of finding a longest VP from the same graph. For a given spatial join operation, once its clusters have been generated, the problem of finding a best scheduler is converted to the problem of finding an MO order from the corresponding CO graph. From Property 3, there does not exist any polynomial time algorithm to find an MO order from the CO graph. A *maximum spanning tree* (MST) based heuristic was proposed in [7] to produce an approximation to MO (AMO) order. We propose to apply the AVP heuristic to find an AMO to guide the spatial cluster scheduling. We have conducted a series of experiments to compare the quality of the AMO order generated by MST and AVP heuristic.

4.4. Experimental evaluation

The experiments were conducted to demonstrate the reduction of the I/O costs in spatial join processing by using the AMO orders to guide the scheduling of processing of clustered join operations.

Table 1: Results of experiment with 10 clusters

Edge	MST	AVP	AVP over MST
10	4721	5570	17.98%
15	3869	4302	11.19%
20	5596	6648	18.80%
22	5416	6536	20.68%
25	5624	7144	27.03%
30	5774	6532	13.13%
33	5575	6300	13.00%
35	6542	7882	20.48%
40	6686	7548	12.89%
45	6944	7910	13.91%
Average	5674.7	6637.2	16.91%

We compare the quality of the AMO orders generated by two methods in term of the overlapping weight of the AMO order. The new cluster sequencing method (i.e., AVP) is simulated against *MST* [7]. In the experiments, most spatial datasets are generated while a small portion of datasets is from real spatial applications. The object sizes change from tens to hundreds of vertices. At each simulation point, the simulation runs 10 times. Since every object needs to be fetched into the memory for the spatial join operation, for simplicity, we measure the I/O cost in terms of the total size of the overlapping objects that are fetched repeatedly into the memory for processing (i.e., the value of the second part in formula (2)).

Table 1 shows the experiment results with ten clusters/vertices in the CO graphs. There were ten experiments conducted with a different number of edges connecting the clusters. For example, for ten edges, the total overlapping weights produced by MST and AVP methods are 4721 and 5570, respectively. Thus, AVP outperforms MST by 17.98%. The average result showed that AVP method can potentially produce 16.91% more total overlapping weight when comparing to MST.

Table 2: Summary of experiment results

Number of cluster	**MST**	**AVP**	**AVP over MST**
10	5674.7	6637.2	16.91%
20	9509.8	11038.9	16.18%
30	14068.9	15708.7	12.11%
40	17592.4	19819	13.28%
50	21539.5	24431.6	14.09%
60	31011.8	34424.5	11.05%
70	38011	42448.2	12.28%
80	38738.5	42870.6	10.69%
90	41586.7	45569.5	9.78%
100	43001.9	48517.4	12.95%
Average			12.93%

Table 2 shows the summary result of the experiments. For each cluster number, we conducted ten experiments and the average results are shown in the table. For example, for ten clusters, the average of total overlapping weight produced by MST and AVP method are 5674.7 and 6637.2, respectively, and the average percentage of performance comparison for each method is also shown in the table (detailed experiments are omitted here).

5. Conclusion

While many optimization applications can be modeled by graphs, these applications are hardly converted directly to a Hamilton path problem because a graph may not contain a Hamilton path, and there is not a sufficient and necessary condition to determine whether or not there exists a solution to the Hamilton path problem of a graph. In addition, the algorithm of finding a Hamilton path from a graph is *NP*-complete. These properties have greatly limited the application to the Hamilton path problem.

We defined a type of variation of the Hamilton Path problem that can be applied to a type of applications, and it always has a solution. We demonstrated that the variation is also a generalization of the Hamilton Path problem in the trivial graph case. As the problem of finding a solution to the variation of the Hamilton Path problem is NP-completed, we developed a heuristic to find approximate solutions to the problem. An application scenario is described to showcase the application potentials of the variation of the Hamilton Path problem. Experiments have demonstrated that the heuristic is better than the existing algorithm that was used to solve the same application scenario in scheduling spatial join operations in spatial databases.

References

[1] Abel, D. (1989). SIRO-DBMS: A Database Tool Kit for Geographical Information Systems. International. *J. of Geographical Information Systems*, Vol. 3, No. 2, pp.103-116.

[2] Abel, D., Gaede, V., Power, R. and Zhou, X (1997), Resequencing and Clustering to Improve the Performance of Spatial Join. *Technical Report*, CSIRO Mathematical and Information Sciences, Australia.

[3] Cook, W. J., & Rohe, A. (1999). Computing Minimum-Weight Perfect Matchings. *INFORMS journal on computing*, 11(2), 138.

[4] Edmonds, J. (1965). Path, Tree, and Flower. *Journal of Math*, 17, 449-467.

[5] Lam, C. P., Xiao, J. and Li, H (2007). Ant Colony Optimisation for Generation of Conformance Testing Sequences using Characterising Sequences, *Proceedings of The 3rd IASTED International Conference on Advances in Computer Science and Technology (ACS2007)*, Phuket, Thailand. ACTA Press. PP140-146.

[6] Lawler, E. L (1976), *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York.

[7] Xiao, J., Zhang, Y. , Jia, X. and Zhou, X. (2000), A Schedule of Join Operations to Reduce I/O Cost in Spatial Database Systems, *Data & Knowledge Engineering*, Elsevier Science B.V, Vol. 35, pp299-317.

[8] Zhou, X., Abel, D. and Truffet, D. (1998), Data Partitioning for Parallel Spatial Join Processing. *GeoInformatica* 2:2, Kluwer Academic Publisher, pp175-204.