Solving The Sequential Ordering Problem With Automatically Generated Lower Bounds

István T. Hernádvölgyi
University of Ottawa
School of Information Technology & Engineering
Ottawa, Ontario, K1N 6N5, Canada
Email: istvan@site.uottawa.ca

August 25, 2003

Abstract

The Sequential Ordering Problem (SOP) is a version of the Asymmetric Traveling Salesman Problem (ATSP) where precedence constraints on the vertices must be observed. The SOP has many real life applications and it has proved to be a challenge as there are SOPs of order 40-50 vertices which have not yet been solved optimally with significant computational effort. We use a novel branch&bound search algorithm with lower bounds obtained from homomorphic abstractions of the original state space. Our method is asymptotically optimal. In one instance, it has proved a solution value to be optimal for an open problem while it also has matched best known solutions quickly for many unsolved problems from the `TSPLIB`. Our method of deriving lower bounds is general and applies to other variants of constrained ATSPs as well.

1 Introduction

The Sequential Ordering Problem (SOP) is stated as follows. Given a graph G, with n vertices and directed weighted edges with the start and terminal vertices designated. Find a minimal cost Hamiltonian path from the start vertex to the terminal vertex which also observes precedence constraints. An instance of a SOP can be defined by an $n \times n$ cost matrix C, where the entry $C_{i,j}$ is the cost of the edge $i - j$ in G, or it is -1 to represent the constraint that vertex j must precede vertex i in the solution path.

The SOP is a model for many real life applications, ranging from helicopter routing between oil rigs [14] to scheduling on-line stacker cranes in an automated warehouse [1].

While Traveling Salesman Problems (TSP) of large order can be solved optimally, there are still unsolved problems of SOPs with 40-50 vertices. Most asymptotically optimal solvers model the SOP as an Integer Program. Unfortunately the exact structure of the SOP polytope is not yet fully understood and therefore these methods achieved only limited success. Our approach is state space search. The partial completions of feasible tours form a directed acyclic graph where the minimal cost path from the start state (the tour with only the start vertex) to the goal state (the tour with all vertices) corresponds to the optimal solution to the SOP. We use automatically generated lower

bounds to prune branches from the search tree. These lower bounds are derived from abstractions of the original state space and correspond to optimal tour completion costs in the abstract space.

Using our technique, we were able to prove a new optimal solution to an unsolved instance in the `TSPLIB` [16] and we were also able to match best known solutions to many so far unsolved instances. The abstraction mechanism described in this paper is general and it is applicable to other versions of constrained ATSPs as well.

2 Related Work

Optimal solutions to some instances of the SOP were obtained by Ascheuer *et. al.* [2] who used a cutting plane approach. The SOP is modeled as an Integer Programming problem derived from the Asymmetric Traveling Salesman Problem (ATSP) polytope [8]. The challenge was to find valid facet-including inequalities that represent the additional precedence constraints. They also employed heuristic tour improvements to derive solutions. Escudero *et al.* [7] used a similar approach but the lower bounds were obtained by Lagrangian relaxation.

The HAS-SOP system of Gambardella *et al.* [10, 9] is a metaheuristic technique. In many instances they obtained the best known upper bounds to unsolved instances. This approach is a form of stochastic search and therefore optimal solutions cannot be guaranteed, however these solutions were obtained very quickly. Similar results using genetic algorithms were achieved by Seo *et al.* [15].

Our lower bounds are actually derived from a state space graph and correspond to an underestimate of tour completion over the remaining vertices. In this sense our bounds are closer in spirit to the well known Minimum Weight Spanning Tree (MST) bound of the TSP or to the Assignment Problem (AP) lower bound of the ATSP. These bounds, however, cannot be easily adopted to incorporate precedence constraints. Christofides *et al.* [3] considered state space relaxations first to generate lower bounds. They also used a state representation very close to ours. The lower bounds correspond to the completion of a relaxation of the original dynamic programming recursion of the SOP. This approach was considered by Mingozzi *et al.* [13] for the TSP with time windows and precedence constraints.

We use large look-up tables of lower bounds which are derived from an abstraction of the state space. These look-up tables are referred to as *pattern databases* because the abstraction corresponds to merging states of the original state space according to some syntactic *pattern*. This technique was invented by Culberson and Schaeffer [4] and was later used by Korf [12] to obtain optimal solutions to the Rubik's Cube for the first time. Edelkamp [6] also used pattern databases to derive optimal plans for STRIPS problems.

3 State Space

The SOP of n vertices is usually defined by an $n \times n$ matrix C of edge costs where $C_{i,j} = -1$ if vertex j must precede vertex i in the tour. Without loss of generality, we label the vertices from 0 to $n-1$; the tour starts at vertex 0 and ends at vertex $n-1$. We write $C[i][j]$ to represent the edge cost $i-j$. Figure 1 shows the `TSPLIB` [16] representation of a 6 vertex SOP. The `EDGE_WEIGHT_SECTION` is an explicit 6×6 cost matrix. For example, the cost of the $0-1$ edge is 2 and the $2-1$ edge is 1. Clearly, start vertex 0 must precede all other vertices while the terminal vertex 5 is preceded by all vertices

```
NAME: Ex6
TYPE: SOP
COMMENT: Example
DIMENSION: 6
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
6
    0    2    1    4    4 1000000
   -1    0    1    4    2    1
   -1    1    0    3    3    0
   -1   -1    3    0    2    3
   -1    2    1    3    0    1
   -1   -1   -1   -1   -1    0
EOF
```

Figure 1: The example SOP $Ex6$ in TSPLIB format

but itself. Since the SOP can visit each vertex only once, the values in the diagonal are irrelevant. The edge from the start to the terminal vertex cannot be taken either[1]. The -1 entry for edge 3-1 indicates that vertex 1 must precede vertex 3. Figure 2 depicts the precedence graph P corresponding

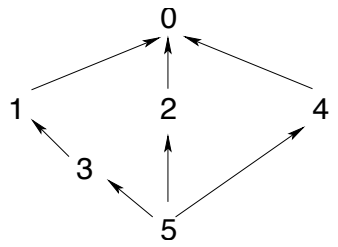

Figure 2: Precedence graph P corresponding to SOP $Ex6$

to this example SOP. There is a directed edge from vertex v_2 to v_1 in P if v_1 has to precede v_2; implied precedences are removed. This graph can be efficiently computed using dynamic programming and it can be used to remove redundant edges from the cost matrix. If there is no direct edge in P from v_1 to v_2, then there should be no entry for edge $v_2 - v_1$ in the cost matrix. For example, there is no edge from 3 to 0 because vertex 1 in any valid tour must be between them. Similarly, the $1-5$ and $0-5$ edges are also deleted from the cost matrix C. We use the $\times$ symbol to mark edges removed.

$$C = \begin{pmatrix} \times & 2 & 1 & \times & 4 & \times \\ -1 & \times & 1 & 4 & 2 & \times \\ -1 & 1 & \times & 3 & 3 & 0 \\ -1 & -1 & 3 & \times & 2 & 3 \\ -1 & 2 & 1 & 3 & \times & 1 \\ -1 & -1 & -1 & -1 & -1 & \times \end{pmatrix} \tag{1}$$

[1]It is customary to put a value for this edge that exceeds the value of any feasible solution

In our representation, a SOP state s corresponds to a partial completion of the tour. It records the current last vertex in this partial tour as well as the vertices which have not been reached yet. This is very similar to the state representation of Christofides *et al.* [3] who instead kept track of the vertices visited so far in the partial tour. For example, consider the intermediate state below for our 6 vertex SOP:

$$s = (3, 5)[2]$$

The tour is currently in vertex 2 and vertices 3 and 5 are yet to be reached. From this information we also know that vertices 0, 1 and 4 are already part of this partial tour but their actual respective order is not kept in the state. The SOP state space S, with this representation is a lattice with the start state at the apex and the goal state at the bottom. The lattice has n levels corresponding to the n vertices in the tour. The start state is at level 0. The state space lattice corresponding to our example $Ex6$ is depicted in Figure 3.

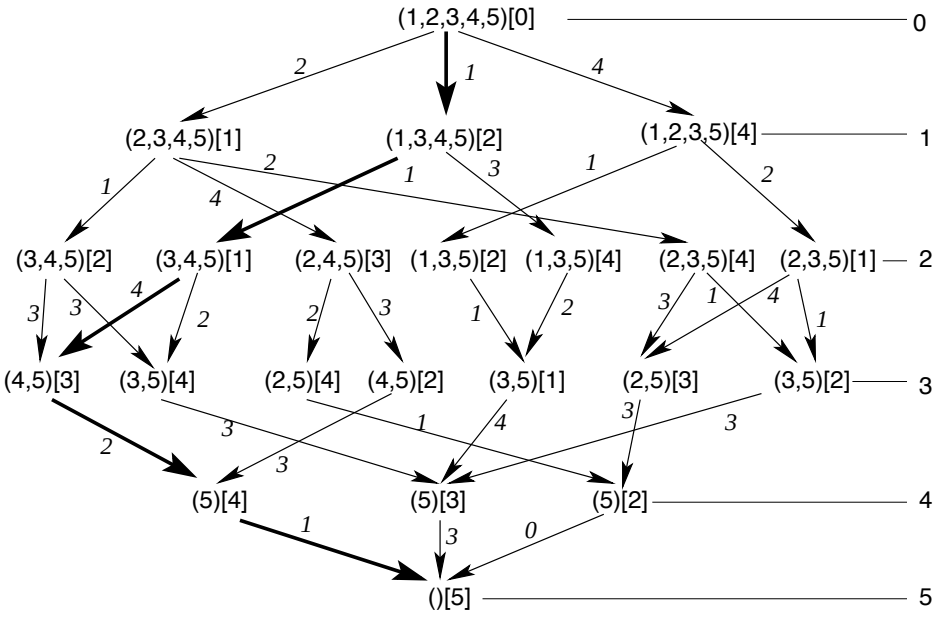

Figure 3: State Space Lattice S

The full lattice not considering precedences has

$$|S(n)| = \left(\sum_{i=0}^{n-3} \binom{n-2}{i} (n-2-i) \right) + 2 \tag{2}$$

states. At level $1 \leq l \leq n-2$, $n-2-l$ vertices are available from the $n-2$ possible vertices, since vertices 0 and $n-1$ are reserved. The last visited vertex in the partial tour could be any of the not chosen l vertices from the $n-2$. Setting $i = n-2-l$, equation 2 follows. There are at most $(n-2)!$ complete tours through S; some are infeasible because of the precedence constraints. $|S(n)|$ is significantly smaller than $(n-2)!$. For example, for a 30 vertex SOP,

$$\begin{array}{lcll} |S(30)| & = & 3,758,096,386 & // \text{ states in } S \\ 28! & = & 304,888,344,611,713,860,501,504,000,000 & // \text{ paths through } S \end{array}$$

The minimum cost path observing precedences is the optimal solution to the SOP. In our case, it is the tour 0-2-1-3-4-5 with cost 9 (Figure 3).

The state space lattice for a SOP of n vertices has n distinct levels. We will write S_i to refer to the states of S on level i. Edges in S are only connecting states of S_i to S_{i+1}. The state representation implicitly encodes level information: the number of vertices available subtracted from $n-1$ is the level where the state belongs. For example,

$$l(\ (2,4,5)[3]\ ) = (6-1) - 3 = 2$$

In the discussions that follow, we rely on this level property of S.

4 Lower Bounds

Let s be a partial tour in the state space lattice S. Let $c(s)$ be the cost of this partial tour and let $b(s)$ be an underestimate of the cost of completing s. Let $c(t)$ be the cost of the best cost full tour t known so far. If $c(s) + b(s) \geq c(t)$ then s can be eliminated as a candidate for an optimal solution. Our search procedure (which we will describe in detail later) uses such lower bounds to prune the space lattice.

The lower bounds will be derived from an abstraction of the state space lattice and stored in a large look-up table we call the *pattern database*. The abstract state space S' is also a lattice with the same number of levels as S. However, $|S'| < |S|$, so we can enumerate it efficiently. The abstract lattice is obtained by clustering states of S on the same level. Figure 4 illustrates the conceptual relationship

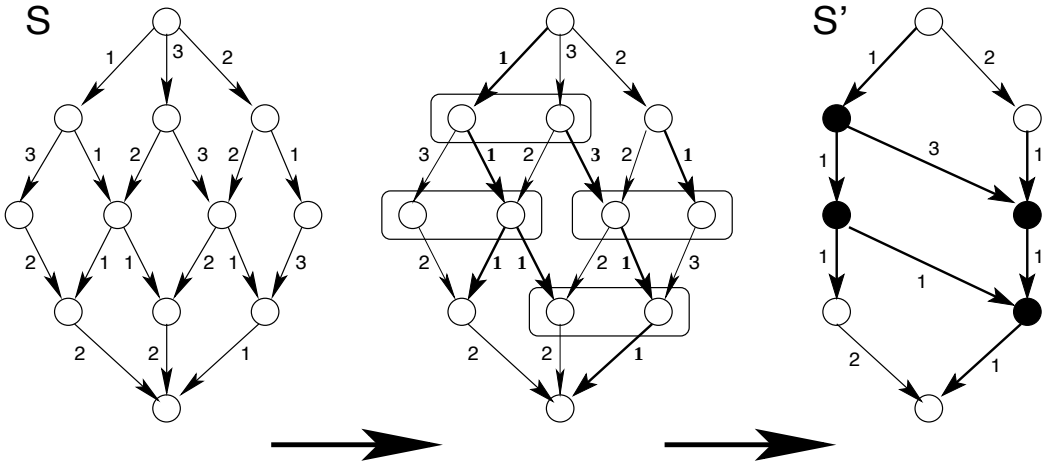

Figure 4: State Space Lattice Abstraction

of the original and abstract lattices. S is the original lattice and S' is the resulting abstraction. The cheapest cost path from the top to the bottom vertex corresponds to the optimal tour in the underlying SOP. We chose some states on the same levels to cluster and identified the cheapest cost edges entering and leaving these clusters (drawn with bold edges). These edges will be retained to connect the clustered states which are now replaced by a single vertex (shown as filled circles) in S'. We will refer to the vertices of S' as abstract states. Since we always chose the cheapest edges, reaching

the bottom vertex from an abstract state s' in S' cannot be more expensive than reaching the bottom vertex in S from any of the original preimages of s'. Therefore the cost of the optimal completion of s' in S' is a lower bound on the optimal completion cost of the preimage states of s' in S. This is exactly what our lower bounds correspond to. However, considering the number of states in the SOP lattice (Equation 2), enumeration of S is infeasible. We have to compute S' without computing S.

Let S_i and S'_i denote the states at level i in S and S' respectively. The level property of the lattice is preserved in S'. The map $\phi_i : S_i \rightarrow S'_i$ is an equivalence relation for $s_i \in S$ where s'_i are the equivalence classes. In essence the action of ϕ_i is to designate the clusters of S_i. $\Phi =< \phi_0, \phi_1, ..., \phi_{n-1} >$ is a vector of such maps with one abstraction for each level. The states of $S' = \Phi(S)$ are obtained by applying the maps in this vector level-wise to S.

The action of ϕ is purely a syntactic relabeling of the state representation. We distinguish between the vertex and the label that refers to it. Initially all vertices have their own label. The start vertex has label 0 and the terminal vertex has label $n-1$ in the n vertex SOP. Our abstractions operate on the set of labels; in the abstract problem there could be more than one vertex with the same label. This does not mean that vertices get "physically" merged, but they are indistinguishable in the abstract problem. For example consider a 5 vertex SOP as shown in Figure 5. For now we omit precedence constraints and edge weights. In the original SOP, each vertex has its own label. The number of tours is 6

$$0-1-2-3-4$$
$$0-1-3-2-4$$
$$0-2-1-3-4$$
$$0-2-3-1-4$$
$$0-3-1-2-4$$
$$0-3-2-1-4$$

Now, let us relabel the vertices with labels 2 and 3 so they now have the same label x. There are only three tours in this abstract SOP:

$$0-1-x-x-4$$
$$0-x-1-x-4$$
$$0-x-x-1-4$$

Note that both vertices with label x are present in each tour. The original cost matrix is:

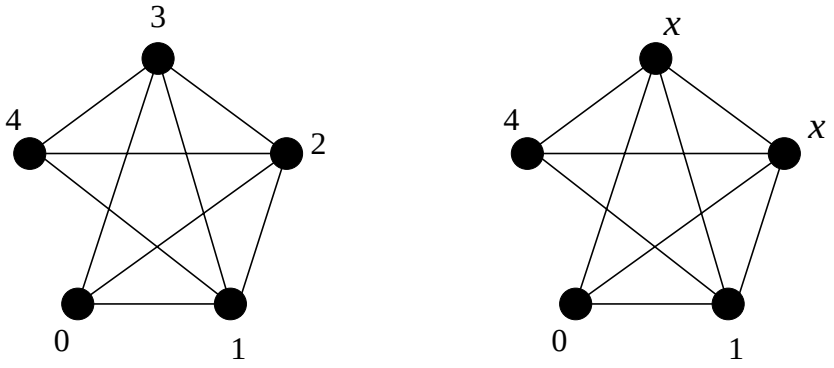

Figure 5: Label abstraction in a 5 vertex SOP

$$C = \begin{pmatrix} \times & c_{0,1} & c_{0,2} & c_{0,3} & \times \\ -1 & \times & c_{1,2} & c_{1,3} & c_{1,4} \\ -1 & c_{2,1} & \times & c_{2,3} & c_{2,4} \\ -1 & c_{3,1} & c_{3,2} & \times & c_{3,4} \\ -1 & -1 & -1 & -1 & \times \end{pmatrix}$$

Since we cannot tell the vertices with label x from each other, the abstract cost matrix has only one entry for edges leading to and leaving x:

$$C' = \begin{pmatrix} \times & c_{0,1} & \boxed{c_{0,2} \quad c_{0,3}} & \times \\ -1 & \times & \boxed{c_{1,2} \quad c_{1,3}} & c_{1,4} \\ \boxed{-1 \\ -1} & \boxed{c_{2,1} \\ c_{3,1}} & \boxed{\times & c_{2,3} \\ c_{3,2} & \times} & \boxed{c_{2,4} \\ c_{3,4}} \\ -1 & -1 & \boxed{-1 \quad -1} & \times \end{pmatrix} = \begin{pmatrix} \times & c_{0,1} & c'_{0,x} & \times \\ -1 & \times & c'_{1,x} & c_{1,4} \\ -1 & c'_{x,1} & c'_{x,x} & c'_{x,4} \\ -1 & -1 & -1 & \times \end{pmatrix}$$

Note that there is a diagonal entry $c'_{x,x}$ since there are abstract paths which include the $x-x$ edge. The abstraction of vertex labels does not reduce the number of vertices but it merges edges and therefore compresses the cost matrix. For example, the $c'_{x,1}$ edge in C' replaces the $c_{2,1}$ and $c_{3,1}$ edges in C.

Definition: *Domain Abstraction*

> Let $D = \{v_1, v_2, ..., v_n\}$ be the set of original vertex labels. Any onto map $\phi : D \to D'$ with $D' = \{v'_1, v'_2, ..., v'_k\}$ where $k \leq n$ is a *domain abstraction*.

We use domain abstractions to create the abstract states. The action of a domain abstraction on a state is to relabel each vertex in the representation. Without loss of generality, let $D = \{0, 1, ..., n-1\}$ and $D' = \{0, 1, ..., k-1\}$; this allows us to use the vertex labels as indices of the cost matrix. We further require that that our domain abstractions preserve the identities of the start and terminal vertices: $\phi(0) = 0$, $\phi(n-1) = k-1$ and no other vertex maps to 0 or $k-1$ under ϕ.

In the example problem $Ex6$, there are 6 vertices. Originally they all have different labels 0, 1, ..., 5. After applying ϕ some of these vertices will be assigned different labels but there are still 6 vertices.

For example, for $n = 6$ and $k = 4$,

$$\phi_2(v) = \begin{cases} 0 & \text{if } v = 0 \\ 1 & \text{if } v \in \{1, 2, 4\} \\ 2 & \text{if } v = 3 \\ 3 & \text{if } v = 5 \end{cases} \tag{3}$$

is an example of a domain abstraction. It is marked ϕ_2 because we will apply this map to level 2 of S.

For example, the states of S_2 (Figure 3) under ϕ_2 fall into the equivalence classes in Table 1. S'_2 has two states: $(1, 2, 3)[1]$ and $(1, 1, 3)[2]$. Note that as a result of this clustering, the original vertices 1, 2 and 4 at level 2 are indistinguishable. We would like to caution that the action of ϕ is not to merge vertices into one but to make them indistinguishable. In state $(1, 1, 3)[2]$ there are two 1's; their original identity is lost. In fact, a tour in S' must contain three 1's as all vertices must be visited exactly once.

$S'_2 \rightarrow$	$(1,2,3)[1]$	$(1,1,3)[2]$
	$\uparrow$	$\uparrow$
$S_2 \rightarrow$	$(3,4,5)[1]$ $(1,3,5)[2]$ $(3,4,5)[2]$ $(1,3,5)[4]$ $(2,3,5)[1]$ $(2,3,5)[4]$	$(2,4,5)[3]$

Table 1: The abstract states $\phi_2(S_2)$

The action of ϕ is fairly trivial on the states. To complete the abstraction of Φ, we also need to know how the edges of the lattice are effected. In other words, the action of Φ on C.

Since $\phi_i \in \Phi$ could all have different number of abstract vertex labels,

$$\Phi(C) =< C'_{0,1}, C'_{1,2}, ..., C'_{n-2,n-1} >$$

is a vector of cost matrices where $C'_{i,i+1}$ represents the connectivity between levels i and $i+1$ in S'.

In the lattices of S and S', all edges connect consecutive levels only, therefore we only consider a pair of adjacent levels at a time. To continue with our example, we define another domain abstraction for level 3:

$$\phi_3(v) = \begin{cases} 0 & \text{if } v = 0 \\ 1 & \text{if } v = 1 \\ 2 & \text{if } v \in \{2,3\} \\ 3 & \text{if } v = 4 \\ 4 & \text{if } v = 5 \end{cases} \tag{4}$$

The states of S_3 under ϕ_3 are listed in Table 2. We now proceed to determine the edges and their

$S'_3 \rightarrow$	$(2,4)[1]$	$(2,4)[2]$	$(2,4)[3]$	$(3,4)[2]$
	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
$S_3 \rightarrow$	$(3,5)[1]$	$(2,5)[3]$ $(3,5)[2]$	$(3,5)[4]$ $(2,5)[4]$	$(4,5)[3]$ $(4,5)[2]$

Table 2: The abstract states $\phi_3(S_3)$

costs between S'_2 and S'_3.

There are 4 labels on level 2 and 5 labels on level 3, therefore the cost matrix we are looking for is 4×5. The rows correspond to the vertex labels of S'_2 and the columns to those of S'_3. This cost matrix, $C'_{2,3}$, is derived from the original cost matrix. Equation 5 shows the values of the original matrix C (Equation 1) clustered and rearranged according to ϕ_2 and ϕ_3. For example, the preimages of 1 under ϕ_2 are 1, 2 and 4. Therefore we take the rows corresponding to these vertices from C and make it the row corresponding to their image (namely 1) in $C'_{2,3}$. The boxed entries correspond to edges which are merged by the abstraction. For example, let us single out the entry in the second row and third column. The second row represents the preimages $\phi_2^{-1}(1)$, which are 1, 2 and 4, while the third row corresponds to the preimages $\phi_3^{-1}(2)$ which are 2 and 3. The 6 values are the entries corresponding to edges $1-2$, $1-3$, $2-2$, $2-3$, $4-2$ and $4-3$ from the original matrix C (Equation 1). In the

abstract space, all these edges will be represented by a single edge.

$$
C'_{2,3} = \begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\phi_3 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 & 0 & 1 & 2 & 2 & 3 & 4 \\
\hline
\phi_2 & & & & & & \\
0 \to 0 & \times & 2 & \boxed{\mathbf{1}} & \times & 4 & \times \\
1 \to 1 & \boxed{\mathbf{-1}} & \boxed{\times} & \boxed{\mathbf{1}} & 4 & \boxed{\mathbf{2}} & \boxed{\times} \\
2 \to 1 & \boxed{-1} & \boxed{\mathbf{1}} & \boxed{\times} & 3 & \boxed{3} & \boxed{\mathbf{0}} \\
4 \to 1 & \boxed{-1} & \boxed{2} & \boxed{1} & 3 & \boxed{\times} & \boxed{1} \\
3 \to 2 & -1 & -1 & \boxed{\mathbf{3}} & \times & 2 & 3 \\
5 \to 3 & -1 & -1 & \boxed{\mathbf{-1}} & -1 & -1 & \times
\end{array}
\Longrightarrow
\begin{pmatrix}
\times & 2 & \mathbf{1} & 4 & \times \\
\mathbf{-1} & \mathbf{1} & \mathbf{1} & \mathbf{2} & \mathbf{0} \\
-1 & -1 & \mathbf{3} & 2 & 3 \\
-1 & -1 & \mathbf{-1} & -1 & \times
\end{pmatrix}
\tag{5}
$$

The question is, what cost should we assign to this edge? As we have hinted before, we are building this abstract space to derive lower bounds. Therefore we will take the most conservative choice. First we introduce some notation. Let l and $l+1$ be the indices of adjacent levels. i' and j' are abstract labels with preimages $P_l(i') = \phi_l^{-1}(i')$ and $P_{l+1}(j') = \phi_{l+1}^{-1}(j')$ respectively. $P_l(i')$ and $P_{l+1}(j')$ are the sets of vertices which get mapped to i' in S'_l and to j' in S'_{l+1}. The entry $C'[i'][j']$ is the single edge $i' - j'$ merging the set of edges $C[P_l(i')][P_{l+1}(j')]$. In our example, for $i' = 1$, $j' = 2$, $P_2(1) = \{1, 2, 4\}$ and $P_3(2) = \{2, 3\}$, while

$$
C[P_2(1)][P_3(2)] = C[\{1,2,4\}][\{2,3\}] = \begin{array}{|cc|}
\hline
1 & 4 \\
\times & 3 \\
1 & 3 \\
\hline
\end{array}
$$

We apply the following rules of assigning edge costs to $C'[i'][j']$:

1. if all edge costs $C[P_l(i')][P_{l+1}(j')]$ are -1 then $C'[i'][j'] = -1$

   The entry -1 represents a precedence constraint, $C_{i,j} = -1$ if j must precede i. Therefore all vertices $P_{l+1}(j')$ precede all vertices $P_l(i')$ in S and i' will precede j' in S'.

2. if all edge costs $C[P_l(i')][P_{l+1}(j')]$ are $\times$ then $C'[i'][j'] = \times$

   This means that there are no edges from any of the preimages of i' to any of the preimages of j' and therefore there should be no edge from i' to j' in S'.

3. if some entries of $C[P_l(i')][P_{l+1}(j')]$ are -1 and some are marked $\times$ then $C'[i'][j'] = \times$

   In this case we know that there are no edges from any of the vertices of $P_l(i')$ to any of $P_{l+1}(j')$ in S but we cannot preserve the precedence constraints as it is not true that *all* of $P_{l+1}(j')$ precedes *all* of $P_l(i')$.

4. if there is at least one non -1 and non $\times$ entry within $C[P_l(i')][P_{l+1}(j')]$ then $C'[i'][j'] = \min(C[P_l(i')][P_{l+1}(j')])$ excluding the -1 and $\times$ entries

   Since there is at least one edge from the set of vertices $P_l(i')$ to the set of vertices $P_{l+1}(j')$, we must preserve this link. We choose the minimum of the non -1 and non $\times$ entries.

We applied these rules to obtain the cost matrix $C'_{2,3}$ in Equation 5.

Theorem 1

Let S be the lattice of an n vertex SOP with cost matrix C. Let $\Phi =< \phi_0, ..., \phi_{n-1} >$ be a vector of domain abstractions as defined above. Let s' be a state in $S' = \Phi(S)$ at level l.

The optimal cost of completing $s' \in S'$ is a lower bound on the optimal completion cost of $s \in S$.

Proof.

Suppose s is at level l in S. Let $p =< v_l, v_{l+1}, ..., v_{n-1} >$ be any precedence observing completion of s. The cost of this completion is

$$c(p) = \sum_{i=l}^{n-2} C[v_i][v_{i+1}]$$

We show that the path $p' =< \phi_l(v_l), \phi_{l+1}(v_{l+1}), ..., \phi_{n-1}(v_{n-1}) >$ in S' is a completion of s' with cost $c(p') \leq c(p)$.

Since p is a feasible path, we know that the entries $C[v_i][v_{i+1}]$ are non -1 and non-$\times$. The cost of this path is

$$c(p') = \sum_{i=l}^{n-2} C'_{i,i+1}[\phi_i(v_i)][\phi_{i+1}(v_{i+1})]$$

While we do not know the exact values of $C'_{i,i+1}[\phi_i(v_i)][\phi_{i+1}(v_{i+1})]$, we know that they were created by applying rule 4 only, which chooses the cheapest of the edge costs between the preimage vertex clusters. The costs $C[v_i][v_{i+1}]$ belong to these clusters, so $c(p') \leq c(p)$.

We still have to prove that there are no precedence constraints in $C'_{i,i+1}$ which would render p' infeasible. For p' not to exist, there must be a precedence constraint for some $0 \leq i \leq n-2$ which prohibits taking the $\phi_i(v_i) - \phi_{i+1}(v_{i+1})$ edge in p'; namely $C'_{k,k+1}[\phi_k(v_{i+1})][\phi_{k+1}(v_i)] = -1$, where $l \leq k \leq n-2$. Only rule 1 introduces precedence constraints into $C'_{i,i+1}$. If there is such an entry, then all of $\phi_k^{-1}(v_{i+1})$ must precede all of $\phi_{k+1}^{-1}(v_i)$; the precondition for rule 1. But this is a contradiction because in p v_i precedes v_{i+1} and p is a precedence observing completion by assumption.

□

As we stressed before, the enumeration of the states of S is infeasible and therefore we have to build S' without knowing S. S' will be enumerated entirely and therefore it has to be relatively small. We expand S' one level at a time from the bottom up by generating the predecessor states on the previous levels. We can do this, since the edges of S' have unique inverses whose costs can be readily obtained from the abstract cost matrices. As we expand the levels we also calculate the cheapest cost of reaching each abstract state from the abstract goal state. These are the lower bounds which are stored in look-up tables.

We assume the existence of perfect minimal hash functions $I_l : S'_l \rightarrow \{0, ..., |S'_l| - 1\}$. This hash function operates on its assigned level and its purpose is twofold. First, it is used to keep track of

states visited in the expansion and their best known completion cost. Second, the largest value of this function is the maximum size of the level. We will later use this knowledge to calculate domain abstractions which yield abstract spaces which are small enough to be enumerated explicitly. We later show how to calculate the value $I_l(s')$ efficiently. Let H_l be an array with $|S'_l|$ many entries, where $H_l[I_l(s')]$ stores the value of the optimal completion cost of s' in S'. If s is on level l in S, then

$$b(s) = H_l[I_l(\phi_l(s))] \tag{6}$$

is a lower bound on completing s in S. If s cannot be completed, $b(s) = -1$.

Since S' is a lattice with a level property and all costs are non-negative, we can calculate the optimal costs one level at a time. That is, we can calculate the values of H_l considering only level S'_{l+1}.

For this, we will need to be able to generate the predecessors on S'_l of a state on S'_{l+1}. While we omit the details, this calculation can be performed in constant time in terms of the number of original vertex labels.

Function $prec(s', \phi_{l+1}, \phi_l)$ returns the set of predecessor states of s' in S'_{l+1} and function $curr(s')$ (current) is the label of the last vertex in the partial tour s'. For each state of $s'_{l+1} \in S'_{l+1}$ we generate its predecessors on S'_l. For each predecessor s'_l, we calculate the minimal cost of reaching it via s'_{l+1} from the abstract goal state. This cost is composed of the edge cost stored in the cost matrix $C'_{l,l+1}$ and the minimal cost of reaching s'_{l+1} from the abstract goal state; the value we stored in $H_{l+1}[I_{l+1}(s'_{l+1})]$. We compare this cost to the entry for the predecessor s'_l in H_l. If we found a better cost, then we update the value $H_l(I_l(s'_l))$.

Algorithm 1: *Calculating the sates of S_l and the values of H_l*

Given

S_{l+1}: abstract states on S_{l+1}
$C'_{l,l+1}$: cost matrix between levels l and $l+1$
H_{l+1}: optimal completion costs of states in S_{l+1}
I_l, I_{l+1}: minimal perfect hash functions for S_l and S_{l+1}

1. **for** $i = 0$ **to** $|S_l| - 1$, $H_l[i] = -1$ **end for** // *all values of $H_l[i]$ are initially -1*
2. **for each** $s'_{l+1} \in S'_{l+1}$ // *for each state on S'_{l+1}*
3. **for each** $s'_l \in \text{prec}(s'_{l+1}, \phi_{l+1}, \phi_l)$ // *for each predecessor s'_l of s'_{l+1}*
4. $c = C'_{l,l+1}[\text{curr}(s'_l)][\text{curr}(s'_{l+1})]$ // *cost from the predecessor s'_l to s'_{l+1}*
5. $b = H_{l+1}[I_{l+1}(s'_{l+1})] + c$ // *cost of cheapest path reaching s'_l via s'_{l+1}*
6. $i = I_l(s'_l)$
   // *if there is no value for $H_l[i]$ or a better one is found, update this value*
7. **if** $H_l[i] < -1$ **then** $H_l[i] = b$
8. **else if** $H_l[i] > b$ **then** $H_l[i] = b$
9. **end for**
10. **end for**

Let us walk through this algorithm in the context of our example $Ex6$ with ϕ_2 and ϕ_3. Figure 6 depicts the two levels S'_3 and S'_2. We have already calculated the states on S'_3 and know the cost of the shortest completion of each state: the values of H_3. We show all possible combinations of states, the ones which were not reachable because of precedence constraints are grayed out. The domain abstraction at the

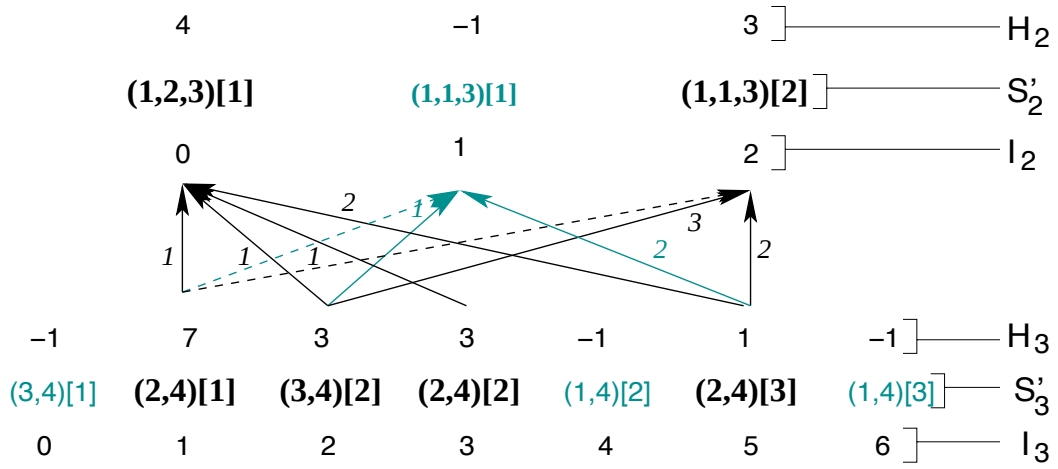

Figure 6: Determining H_2

previous level is ϕ_2 (Equation 3). We also have derived the cost matrix $C'_{2,3}$ (Equation 5) which can be used to obtain the edge costs and the precedence constraints between these levels.

First, $H_2[i]$ is initialized to -1 (*line 1*). We have not calculated the states on S'_2, yet, so initially we mark each as "not reached". Next, we iterate through the reachable states of S'_3 (*line 2*). The first reachable state is $(2,4)[1]$. We show the predecessors in two steps. The predecessors of this state under ϕ_3 are

$$\text{prec}((2,4)[1], \phi_3, \phi_3) = \left\{ \begin{array}{l} (1,2,4)[2] \\ (1,2,4)[3] \end{array} \right\}$$

We need the images of these states under ϕ_2, which are

$$\text{prec}((2,4)[1], \phi_3, \phi_2) = \left\{ \begin{array}{l} (1,2,3)[1] \\ \cancel{(1,1,3)[1]} \\ (1,1,3)[2] \end{array} \right\}$$

The state $(1,1,3)[1]$ is unreachable because a precedence constraint is violated. While we described the action of ϕ_l on S'_l that it clusters states from S'_l, we do not calculate the states of S'_l this way. We build S'_l by generating the predecessors of S'_{l+1}. This is first achieved by a syntactic manipulation of the vertex labels of s'_{l+1}, done in the two stages mentioned above. Then, the states are checked against $C'_{l,l+1}$. The abstractions ϕ_{l+1} and ϕ_l may or may not have preserved the precedence constraints originally defined in C. Fortunately, in this case, this violation can be discovered from $C'_{2,3}$ (Equation 3) as both ϕ_2 and ϕ_3 preserved the precedence constraint. There are cases, however, when the precedence constraint is eliminated by the abstraction (Rules 3 and 4) and predecessors are generated which have no preimages in S. These states have two undesirable effects. First, they could result in poor lower bounds because they create (potentially cheap) paths in S' which do not correspond to feasible paths in S. Second, we have a non-negative value for $H[s']$ so the search tries to complete s while it could have discovered that s has no feasible completion. In Appendix 10.2 we describe how we can check at least for some cases if an abstract state has no preimage in the original state space. Because in this particular case the constraint is preserved,

$$\text{prec}((2,4)[1], \phi_3, \phi_2) = \left\{ \begin{array}{l} (1,2,3)[1] \\ (1,1,3)[2] \end{array} \right\}$$

Now, $C_{2,3}[1][1] = 1$ so the edge connecting $(2,4)[1]$ and $(1,2,3)[1]$ is assigned this value (*line 4*). $C_{2,3}[2][1] = -1$ which indicates that $(2,4)[1]$ is not reachable from $(1,1,3)[2]$; indicated by a dashed arrow in Figure 6. $I_2((1,2,3)[1])) = 0$ (*line 6*) and $H_2[0] = -1$. As we have not reached $H_2[0]$ yet, we set $H_2[0] = H_3[1] + 1 = 8$ (*line 7*). Later this value will change because a cheaper route is discovered to this state. This algorithm is a version of Dijkstra's single-source all shortest-path algorithm [5] adopted to a lattice with a level property with non-negative costs.

There is a large number of domain abstractions for a given SOP. We are interested in those that yield good lower bounds. We will later confirm experimentally the intuitive conjecture that the larger S' the more accurate the lower bounds. Therefore one of our aims is to build the largest S' we can handle within our memory limits. Let $\|S'\|$ denote the size of S' if we ignored precedence constraints (that is, if we generated even those states which represent infeasible partial tours). $\|S'\|$ solely depends on the granularity of domain abstractions. Our total storage requirements for the lower bounds (Algorithm 1) is exactly $\|S'\|$.

Definition: *Granularity*

> Let $\phi : D \rightarrow D'$ be a domain abstraction. The *granularity* of ϕ, $g(\phi)$, is a sorted vector of length $|D'|$ where the elements correspond to the number of preimages of each abstract label in D'.

For example, $g(\phi_2) =< 3,1,1,1 >$, since label 1 has three preimages while the other three labels each have only one. If Φ^1 and Φ^2 have the same granularity domain abstractions for the same levels, then $\|\Phi^1(S)\| = \|\Phi^2(S)\|$. Therefore our second objective is to find a "good" set of domain abstractions Φ within the sets of domain abstractions of the same granularities.

As enumerating S' is an expensive operation, we will not consider generating many possible Φ's. Instead we will use greedy search which climbs to a local optimum according to a user provided evaluation function. Since we build S' bottom up (Algorithm 1), we calculate ϕ_i from S_{i+1}. Initially, we start with ϕ_{n-1} which maps labels to themselves; $\phi_{n-1}(x) = x$ for all $x \in D$. Suppose we are at level $i+1$ in Algorithm 1. This means that we have already calculated levels $i+2, ..., n-1$. We build the perfect hash function I_i for level i using $\phi_i = \phi_{i+1}$. If the maximum value of I_i is larger than some user imposed value, we make ϕ_i more abstract. "More abstract" specifically refers to a domain abstraction which would result in fewer states on the next level. If we could enumerate S_i with ϕ_{i+1}, then $C'_{i+1,i} = C'_{i+1,i+1}$. We then proceed according to Algorithm 1. Now suppose, we discover that ϕ_i has to be made more abstract. There is a very large number of abstractions which are more abstract than ϕ_i, so we restrict ourselves to those which can be obtained by merging two labels of ϕ_{i+1} to a single one in ϕ_i. For example, if ϕ_{i+1} has $k+1$ labels, then ϕ_i will have k labels. We still require that the identities of the start and terminal vertices are kept unique, so we reserve these labels. Therefore we have $\binom{n-3}{2}$ pairs of vertex labels to consider (which is of quadratic order in n). To estimate how good abstraction ϕ_i would be, we calculate $C'_{i+1,i}$ for each pair of labels. Suppose we choose labels v_1 and v_2 to be merged by ϕ_i. $v_1, v_2 \in D_{i+1}$ and suppose they receive the label $v' \in D_i$. The rows

corresponding to v_1 and v_2 in $C'_{i+1,i}$ will be merged according to our rules of edge cost assignments.

$$C'_{i+1,i+1} = \begin{pmatrix} & & & \dots & \\ v_1: & -1 & 3 & \dots & 0 \\ & & & \dots & \\ v_2: & -1 & 2 & \dots & \times \\ & & & \dots & \end{pmatrix} \implies C'_{i+1,i} = \begin{pmatrix} & & \dots & \\ v': & \boxed{\begin{matrix}\mathbf{-1}\\ -1\end{matrix}} & \boxed{\begin{matrix}\mathbf{3}\\ \mathbf{2}\end{matrix}} \dots & \boxed{\begin{matrix}\mathbf{0}\\ \times\end{matrix}} \\ & & \dots & \\ & & \dots & \end{pmatrix} \tag{7}$$

The merge of rows is exemplified by Equation 7. We calculate the row merge for all pairs of labels and choose the pair which minimizes a user provided error measure E_{v_1,v_2}. The ones we have considered in our experiments were the following:

preserve precedences : choose v_1 and v_2 which minimizes the number of precedence constraints that get eliminated by the merge. If there is more than one best candidate, choose the one that minimizes the absolute error or the normalized error (see below).

absolute error : choose v_1 and v_2 which minimizes

$$E(v_1, v_2) = \sum_{j=0}^{|D_{i+1}|-1} \Big| \; C'[v1][j] \quad - \quad C'[v2][j] \; \Big|$$

If $C'[v1][j]$ or $C'[v2][j]$ is -1 or $\times$ then their values the value is taken to be 0. Some user defined penalty could also be applied if a precedence constraint is eliminated.

normalized error : similar to absolute error, but the absolute value of the difference is divided by the larger of the costs.

It is possible that after merging the two chosen labels ϕ_i is still not abstract enough; the number of states at S_i still exceeds our limit. In this case we repeat this merging step as many times as needed until a suitable granularity domain abstraction is found.

For a SOP of n vertices, the lattice has the most states at the middle levels, this is where we need the coarse granularity abstractions. However the upper levels are much smaller than the middle ones, so we could make the domain abstractions less abstract and still expand it within the given limits. Instead of merging rows, we consider splitting them. There are many more combinations of splitting than merging, therefore we again limit the combinations we consider. Let $\phi_{i+1}(v) \in D_{i+1}$ have preimages $P_{i+1}(v) = \phi^{-1}_{i+1}(v) \in D$. We only consider one kind of label split: taking a single vertex label from $P_{i+1}(v)$ and giving it its own label. We choose the pair v_1 and $v_2 \in P_{i+1}(v)$ which *maximizes* our error measure in $C'_{i+1,i}$. In other words, we want the largest discrepancy between the resulting rows v_1 and v_2.

Since we can vary the granularity of abstractions from one level to the next, we are able to enumerate abstract space lattices which maximally utilize available memory.

5 Search

The SOP state space is a lattice which is also a directed acyclic graph (DAG). Because it is acyclic and finite, a systematic depth-first expansion of the space always terminates. Our search procedures

are based on depth-first branching. Since the depth of the solution is fixed at n for the n vertex SOP, the memory requirement for this search is altogether only $O(n^2)$ states (n successor arrays on the stack). Therefore we can practically use all memory to store the lower bounds. Let

$$f(s_l) = c(s_l) + b(s_l) \tag{8}$$

be an estimate of the cost of a tour which is comprised from the cost of the partial tour $c(s_l)$ and its estimated completion cost $b(s_l)$. b is precalculated and obtained by looking up the entry $H_l[I_l(\phi_l(s_l)]$ as we described in the previous section. The space is explored in depth first order. The successors are generated by the function succ(s_l), which uses the cost matrix associated with the SOP to eliminate those partial tours that would violate precedence constraints. The successors are sorted in increasing order according to their f values (Equation 8).

Algorithm 2: *Depth-first Branch&Bound*

Given

s_l: current partial tour on level l
$c(s_l)$: cost of s_l
ub: the value of the best known tour, -1 initially (global variable)

1. *dfbb*(s_l,$c(s_l)$)
2. **if** $l = n - 1$ **then**
3. **if** $c(s_l) < ub$ **or** $ub = -1$ **then**
4. $ub = c(s_l)$ // *best solution so far*
5. **return** // *bottom reached*
6. **for** $s_{l+1} \in \text{sort}_f(\text{succ}(s_l))$ // *successors of* s_l *on* $l+1$
7. **if** $f(s_{l+1}) \geq ub$ **return**
8. *dfbb*(s_{l+1},$c(s_l) + C[s_l][s_{l+1}]$)
9. **end for**

If we reach the terminal vertex, then we check whether this tour has a lower cost than the cost of the best known tour (*line 2*). If it does, we update the value of the cheapest tour. Otherwise, we generate the valid successors and sort them in order of their f values (*line 6*). We consider those partial tours first whose estimated total cost is cheaper. At any point in the loop, if we discover that the cheapest tour under s_{l+1} cannot be cheaper than the cost of the best known tour, we abandon all further successors (*line 7*) as they are known to cost at least as much as the current s_{l+1}. Otherwise the algorithm keeps branching (*line 8*).

This algorithm can be greatly improved for solving the SOP. Since Algorithm 2 branches depth-first, its action is to solve subtrees of increasing depth from the bottom up. When the algorithm is called recursively with a successor (*line 8*), it can be thought of as solving a new search tree whose depth is one level smaller. However in the case of a SOP, this subtree is also a solution to a smaller SOP $\bar{S}$. Therefore if the lower bounds obtained for S do not yield an optimal solution to the subtree within a user specified computational limit, then this subtree can be solved as a smaller individual instance of a SOP with its own lower bounds.

Theorem 2

Let C be the cost matrix of a SOP and let s_l be a state in S with representation

$$s_l = (v_{l+1}, v_{l+2}, ...v_{n-1})[v_l]$$

The vertices $v_0, ..., v_l$ are part of the partial tour represented by s_l. Let

$$\psi : \{v_l, ..., v_{n-1}\} \rightarrow \{0, .., n-l-1\}$$

be an isomorphism which assigns a unique index to v_l and the vertex labels not yet reached by s_l. We only require that $\psi(v_l) = 0$ and $\psi(v_{n-1}) = n - l - 1$. ψ simply established an order of the unvisited vertices and the current last vertex such that the current vertex is the first and the terminal vertex is the last. Let $\psi(C)$ be a matrix with rows and columns chosen by ψ from C. The SOP, $\bar{S}$, defined on the vertices $\{v_l, ..., v_{n-1}\}$ has cost matrix $\bar{C}$ with values of $\psi(C)$ and precedence constraints added to reflect that $\psi(v_l) = 0$ is the start vertex and $\psi(v_{n-1}) = n - l - 1$ is the last vertex. In essence, $\bar{C}$ is the rows and columns of C with indices of the unreached vertices.

The optimal completion cost of s_l in S is the cost of the optimal solution to the SOP $\bar{S}$.

Proof.

The SOP $\bar{S}$ is over $k = n - l$ vertices. Let

$$\bar{p} =< \bar{v}_0, \bar{v}_1, ..., \bar{v}_{k-1} >$$

be the optimal cost tour in $\bar{S}$. This tour corresponds to the completion of s_l

$$p = \psi^{-1}(\bar{p}) =< \psi^{-1}(\bar{v}_0), \psi^{-1}(\bar{v}_1), ..., \psi^{-1}(\bar{v}_{k-1}) >$$

We prove that p is a complete (1), precedence observing (2) and optimal (3) completion of s_l.

1. p is clearly a complete completion of s_l as it includes each unvisited vertex exactly once, starts at vertex v_l (since $\psi^{-1}(\bar{v}_0) = v_l$) and terminates in vertex v_{n-1} (since $\psi^{-1}(\bar{v}_{k-1}) = v_{n-1}$).
2. Suppose there is a constraint in C which states that v_x must precede v_y in S ($l < x, y < n$). Then there is an entry $C[v_y][v_x] = -1$ which is preserved by $\bar{C}[\psi(v_y)][\psi(v_x)] = -1$, so $\psi(v_x)$ precedes $\psi(v_y)$ in $\bar{p}$. Therefore p is precedence observing with respect to C.
3. Suppose p is not an optimal completion of s_l in S. Then there exists another path $q =< u_l, u_{l+1}, ..., u_{n-1} >$ which is cheaper than p. This path has the image

   $$\bar{q} = \psi(q) =< \psi(u_l), \psi(u_{l+1}), ..., \psi(u_{n-1}) >$$

   in $\bar{S}$. Since q is a path, the entries $C[u_i][u_{i+1}]$ for $0 \leq i < n$ are non-negative and so are the entries $\bar{C}[\psi(u_i)][\psi(u_{i+1})]$. Suppose there is a precedence constraint that u_x precedes u_y for $l \leq x, y < n$; ie. $C[u_y][u_x] = -1$. This constraint is present in $\bar{C}$ as $C[\psi(u_y)][\psi(u_x)] = -1$ so $\psi(u_x)$ precedes $\psi(u_y)$ in $\bar{q}$. But $c(\bar{q}) < c(\bar{p})$ which is a contradiction since $\bar{p}$ is assumed to be the optimal tour in $\bar{S}$. Therefore p is an optimal completion of s_l.

□

The number states on level S_l of an n vertex SOP for $l < 0 < n-1$ is

$$|S_l| = \binom{n-2}{l}(n-2-l) \tag{9}$$

From this, it is clear that the middle levels are far larger then the top and bottom ones. When we build the abstract lattice S', we can enumerate some of the bottom layers with abstracting only a few of the original labels and therefore at these levels the lower bounds will be very accurate. However, the middle layers of S are larger and coarser granularity abstractions are needed in order to calculate S'. This will deteriorate the quality of the lower bounds at levels closer to the top. At levels where the lower bounds are inaccurate, Algorithm 2 will not prune as many branches. However, using the property that tour completions are themselves solutions to SOP instances, we can modify Algorithm 2 to solve such subtrees faster. Let L_l be a user specified limit on how many expansions the user is willing to tolerate for solving the subtree at level l with the current lower bound. When this limit is reached, the current search is abandoned and the completion of s_l is solved as an independent SOP. The subproblem SOP has fewer vertices (and therefore it is a much smaller problem) so we expect to generate more accurate lower bounds. In fact, we do not necessarily have to find the actual optimal solution to this subproblem.

Algorithm 3: *Recursive Depth-first Branch&Bound*

Given

C: cost matrix
ub: upper bound for pruning
L: vector of tolerance values

```
01. rdfbb(C,ub,L)
02.     for i = 0 to i < n E[i] = 0 // init nodes expanded counts
03.     dfbb(C,s_0,0)

04. dfbb(C,s_l,c(s_l))
05.     if l = n - 1 then
06.         if c(s_l) < ub or ub = -1 then
07.             ub = c(s_l) // best solution so far
08.         return // bottom reached
09.     inc(E[i])
10.     if E[i] ≥ L[i] then
11.         rdfbb(C̄,ub-c(s_l),L) // solve the subproblem SOP over the remaining vertices
12.         return
13.     for s_{l+1} ∈ sort_f(succ(s_l)) // successors of s_l on l + 1
14.         if f(s_{l+1}) ≥ ub return
15.         dfbb(C,s_{l+1},c(s_l) + C[s_l][s_{l+1}])
16.     end for
```

The additions to Algorithm 2 are the lines 9-12. If the tolerance is exceeded, we generate $\bar{C}$ which is the cost matrix of the subproblem SOP over the remaining vertices and only looking for solutions

which are better than $ub - c(s_l)$. In most cases it takes much less time to prove that no solution exists whose value is less than this new bound than actually finding the optimal solution in the subproblem. Running our initial experiments we also realized, that it is more efficient to check for violations at the levels closer to the top first. If L_2 and L_5 are both exceeded by the current search, then we choose to solve s_2 instead of s_5 because the solution to s_2 also solves s_5. Of course, it is not easy to determine what the optimal limit values should be. The overhead of calculating new lower bounds must also be taken into account. We can accurately estimate the computational effort involved in building the abstract space since we give exact limits on the size of a level we can enumerate. In our experiments we set L_i approximately 10-15 times the size of the abstract space based on a few experiment we ran initially. It would require further experimentation to verify if a good set of limits can be efficiently found.

There is another property which can be used to improve our search. The precedence constraint "v_i must precede v_j" can be rephrased as "v_j must occur after v_i". Regarding the solution cost, the direction of the search is arbitrary. However, it has been observed [11] that the direction of the search is relevant with respect to computational time to find the optimal solution. This is related to the difference in the branching factors: it is possible that a state in S has more branches on average in one direction than in the other. While it is computationally infeasible to determine the branching factor in the SOP as it is decided by the precedence constraints, it can be calculated quickly down to a few levels from the start. When we consider solving a subtree independently, we already commit to significant computational overhead by calculating new lower bounds. With some additional effort, we can also determine how many states there are in the new SOP at say levels 1,2 and 3 in both directions. Based on these values we can choose the direction we expect to take less work, since solving subtrees at these levels take the most computational effort.

Depth-first branch&bound does not keep track of states already reached and therefore some states are reached many times during search. From the representation

$$s_l = (v_{l+1}, v_{l+2}, ..., v_{n-1})[v_l]$$

we know that the vertices $v_0, ..., v_l$ have been used, but we only know that v_0 was the start vertex and that this partial tour ends in v_l. The different precedence observing combinations of $v_1, ..., v_{l-1}$ are all valid partial tours represented by s_l and in the worst case all of them could be traversed in the search. These different paths to v_l are often referred to as transpositions in the search literature. To minimize the number of transpositions, we can utilize *transposition tables* to store the cost of the cheapest cost of reaching s_l so far. If the algorithm encounters s_l with a higher cost, then s_l does not have to be expanded again. The drawback of keeping transposition tables is the memory overhead. Since we use memory to store lower bounds, we would have to consider if it is worth to trade some of it to store transposition tables instead. In the case of the SOP, it turns out that there is a justifiable answer. We already have a perfect hash function I_l which operates on S'_l (see Appendix 10.1). The hash on S_l is just a special case of I_l where ϕ maps each vertex to label itself. Therefore we also have a perfect minimal hash function to index the transposition tables associated with a particular level. There are no transpositions on levels 0, 1 and 2. The size of level l is given by Equation 9. Therefore, we can calculate how much memory we would need exactly to store transpositions at levels 3 and below. Since we want to avoid re-expansions at the higher levels, we can utilize transposition tables for these shallower levels. In our experiments we found it worth-while to use a few levels of transpositions tables.

Depth-first branch&bound finds feasible paths of decreasing cost. Since these costs are used as upper bounds to prune branches, it is very important to find cheap tours early. The lower bounds stored in a pattern database are also used in sorting the branches. As our experiments will show, having tighter lower bounds in a pattern database does not guarantee cheaper initial solutions. Therefore we decided to use a heuristic improvement method also employed by [2, 10, 15]. Whenever a solution is found (Algorithm 2 line 4), we run a 3-opt tour improvement heuristic. We also alternate directions each time a user specified number of node expansion limit is reached. The searches in both directions work on refining the best known solution so far. The rationale of the direction switch can be understood from recursion of depth-first branch&bound. The tail end of the best known tour is much more refined as many more feasible permutations of the vertex labels have been evaluated. If we displayed the sequence of vertex labels being considered during search, we would see that the vertices of the partial tours near the start vertex hardly change while the tail end vertices are shuffled a lot. Changing the direction tries to force changes in the other end of the tour as well.

6 Results

The two control parameters we supply to calculate the lower bounds are the maximum number of abstract states on a single level (maximum width) and the error function which the abstractions between consecutive levels optimize. In this discussion we use the maximum width as the measure of the size of pattern database as it is directly proportional to the total memory requirements (compare the columns "Maximum Width" and "Actual Size" in Table 4). We have considered four different error functions. These are the ones discussed in the previous section, namely "Preserve Precedences", "Absolute Error" and "Normalized Error", and we also included "Random" which picks the labels to be merged randomly. In all of our experiments we found that "Absolute Error" (AE) generates the tightest lower bounds, even when the penalty value for eliminating a precedence constraint is zero. Applying a good penalty value could often tighten the bounds. In one particular experiment we found that minimizing AE clearly outperforms the other measures and in some cases it generates almost 100 times higher lower bounds than randomly chosen abstractions. The lower bound of the start state, b_0, is also a lower bound of the optimal solution. In this experiment, we obtained significantly higher b_0's than the best posted value of 69,569. This problem is referred to as `p43.4` in the `TSPLIB`. Table 3 shows the results of this experiment. b_0 is the lower bound of the start state, ave(b) is the average value of all lower bounds stored and Time is the CPU time in seconds needed to build the patter database. The values in bold are higher than the best lower bound reported for this problem previously. For each size, we generated 20 random pattern databases and took the highest b_0, the highest ave(b) and reported the time for the shortest calculation. It is clear from the results that even the best values chosen from 20 random databases are far worse than the one built optimizing AE. In fact, our smallest AE pattern database has much higher lower bounds than the best of 20 random ones which is also 16,384 larger. While these large discrepancies were not typical in our experiments, our results suggest that experimentation with the error measure for a particular SOP could be well worth the effort. This experiment also reveals an interesting phenomenon that we encountered on more occasions. The AE pattern database with width 2000 is better than many of the larger ones minimizing the same error measure. At this time we have no explanation other than the abstractions generated seem to preserve the costs between the levels particularly well.

We are most interested in how our lower bounds perform in actual searches. We chose a few unsolved problems from the `TSPLIB` and we also created a testbed of 16 smaller problems. First we describe

Max Width	Random			Absolute Error		
	b_0	ave(b)	Time	b_0	ave(b)	Time
500	400	402.90	0.57	53,965	53,704.10	0.57
1,000	435	668.63	0.69	54,110	47,218.00	0.62
2,000	490	830.25	0.80	**80,690**	59,256.60	0.77
4,000	510	848.48	0.85	54,110	50,163.50	0.92
8,000	545	1,219.24	1.00	54,115	52,356.10	1.13
16,000	580	1,631.24	1.15	54,000	50,853.30	1.17
32,000	630	2,146.16	1.46	**80,890**	65,315.20	1.29
64,000	655	2,624.48	1.69	54,045	52,732.60	2.14
128,000	680	2,934.33	2.13	**81,055**	60,713.50	2.24
256,000	700	3,618.05	2.50	**81,140**	77,298.50	2.85
512,000	710	4,925.33	3.30	**81,110**	69,782.70	2.63
1,024,000	955	5,000.84	4.05	**81,320**	74,026.30	3.71
2,048,000	965	7,841.39	5.82	**81,350**	65,885.20	4.48
4,096,000	1,130	8,218.48	8.95	**81,385**	70,395.50	6.23
8,192,000	1,265	9,704.65	18.43	**81,585**	70,473.30	12.87

Table 3: Generating Lower Bounds for `p43.4`

the `TSPLIB` experiments.

We did not manage to improve on the best known upper bounds of the `TSPLIB` instances but in one case we proved that the upper bound of 83,005 for `p43.4` is also optimal[2]. On the other hand, we did match the best reported upper bounds within 60 minutes of CPU time to the open problems `p43.2`, `p43.3`, `p43.4`, `ry48p.2`, `ry48p.3`, `ry48p.4`, `ft53.2` and `ft53.3`. In these experiments we used bi-directional branch&bound and also utilized a 3-opt edge exchange heuristic.

To study the behaviour of our lower bounds, we created a testbed of 16 smaller problems. Each problem is a SOP over 30 vertices. These 16 SOPs are derived from 4 base problems. The first one was obtained by choosing 30 points on a 500×500 integer grid. The edge costs correspond to the rounded Euclidean distances between these points. We also picked one third of the edges randomly and added random noise within ±50% of the edge cost. The second SOP was generated exactly in the same manner but we selected the 30 points on a 6×6 grid. This latter only has edge costs in the interval $[0, 8]$. The edge costs for the third and fourth SOPs were generated by choosing uniform random integers from $[0, 1000]$ and $[0, 10]$ respectively. Next, we generated four random precedence graphs. We applied the constraints implied by these to each of our four base problems and obtained SOPs with 107 (1), 160 (2), 210 (3) and 253 (4) precedence constraints. These include the 57 trivial precedences which are due to the fact that the start and terminal vertices are designated. For ease of reference, we named these 16 test problems suggestively. 500×500-2 is the SOP whose vertices are chosen from the 500×500 grid and to which the second precedence graph was applied. Therefore its cost matrix has 160 precedence constraints.

First, we investigated the memory and search speed trade-off. That is, how increasing memory to store lower bounds effects search speed. To this end, we solved the 500×500-3 SOP instance with 7 different pattern databases of exponentially larger size. For all these pattern databases, we optimized the "Absolute Error" measure in the abstract space. The results of this particular experiment represent

[2]which also contradicts the upper bound posted as 82,960 whose origin we could not trace

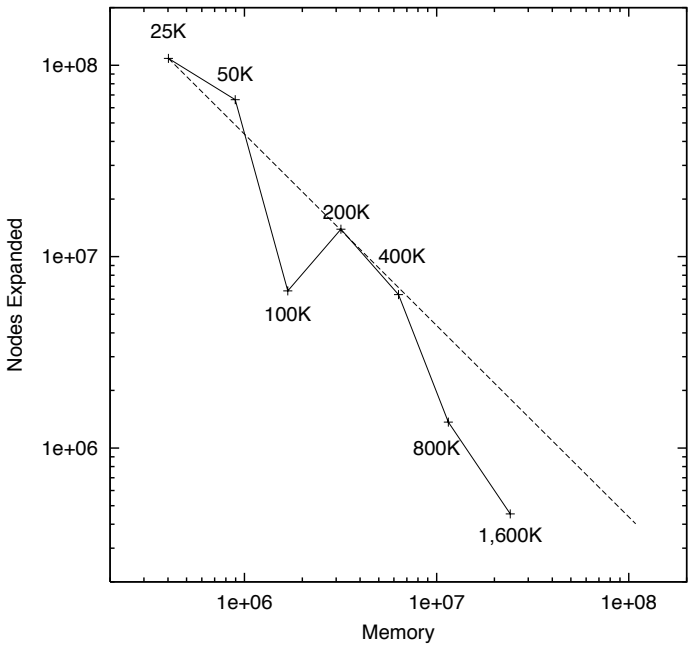

Figure 7: Nodes Expanded vs. Memory

the general trend we observed in the other experiments as well. Figure 7 plots the number of nodes expanded during search and the actual size of the pattern database, this time measured as total number of entries. Both axes are on the logarithmic scale. If the trade-off was 1-1, then the data points would be on a line with slope -1 (drawn as a dashed line). In fact, in this case it seems that the trade-off is even better. Doubling memory to store lower bounds results in less than half of the nodes expanded in the search. The notable exception is the pattern database with width 100K. This one clearly outperforms the one double its size and it is as good as the one four times larger. The detailed statistics of this experiment are listed in Table 4 and Figure 8 plots the solution cost vs. the number of nodes expanded during search. These tell why the 100K pattern database outperforms the one with width 200K. It is clear from Figure 8 that the database with width 200K starts with the costliest solutions a lot of search effort is needed before the cheaper solutions are found. The value of the first solution found also seems to be uncorrelated with the size of the pattern database (Table 4). In this particular case, the smallest pattern database finds the cheapest solution. The lower bounds play two roles in our search algorithm. They are used to prune branches from the search tree as well as ordering the branches. The value of the first solution depends mostly on how the branches are ordered. From the results of this experiment and other experiments we performed we cannot establish any relationship between the size of the pattern database and the quality of the initial sequence of solutions. We did compare to random ordering, which turned out to be consistently worse. However, to date, we found no telling statistics of the pattern database which would suggest cheap initial solutions. In these experiments we did not use the 3-opt heuristic tour improvement so we can study this behaviour. The focus of our currently running experiments is to investigate how we can force cheap solutions early. At this time we have no statistics.

Max Width	**Actual Size**	b_0	**First Solution**	**Nodes Expanded**	**Search** (*sec*)	**DB** (*sec*)
25,000	402,043	2,577	4,183	108,618,722	261.61	0.11
50,000	895,986	2,905	4,655	66,025,147	160.84	0.14
100,000	1,680,769	3,023	5,048	6,621,231	19.16	0.72
200,000	3,178,762	3,065	5,433	13,920,570	40.68	1.33
400,000	6,339,234	2,975	4,183	6,339,234	19.23	2.14
800,000	11,517,316	3,292	4,375	1,367,800	4.54	3.62
1,600,000	24,147,235	3,109	4,695	452,692	1.80	5.10

Table 4: Results of Solving 500×500 3 with Different Databases

While nodes expanded is an appropriate and exact measure to evaluate search performance, we also have to take into account the overhead of building pattern databases. Figure 9 plots the solution cost (y-axis) and the actual CPU time (x-axis, logarithmic scale) that it took altogether to find the solutions. The computational overhead is represented by the time offset of the initial solutions. From this plot we can tell, that even with the overhead, larger pattern databases result in faster searches. The order is almost the same as established by the number of nodes expanded, except that the pattern database with 100K level limit now also outperforms the database whose maximum width is 400K.

We are also interested in how adding more precedence constraints effect search performance. Ascheuer *et al.* [2] observed that their branch&cut method is more effective for SOPs which have less precedence constraints. The limiting factors in branch&cut, besides the dimensionality, are the size of the pool of inequalities that have to be searched and the construction of the facet including inequalities. Non of these are eased by more precedences. For us, more precedence constraints mean less branching in the search space and therefore less node expansions. On the other hand, it is more likely that more of the original constraints would be eliminated by the abstractions if there were more to begin with. Loosing a significant portion of the original constraints could result in poor lower bounds. Our experiments indicate that in most cases the more constrained the SOP the faster our searches. Table 5 shows the results of our experiments with the 16 SOPs. We chose 3 different size pattern databases; with maximum level widths of 50K, 250K and 1,250K states respectively. In all cases we minimized the "Absolute Error" measure when building the abstract spaces. While we can easily solve these problems to optimality with large databases, we also used a small and a moderate size database as well to study how adding more constraints effects the quality of lower bounds with different size pattern databases. From our experiments we can establish that regardless of the size of the pattern database, in general, the more constrained the SOP the earlier the search terminates. The two exceptions are 0-1000-3 and 0-10-3. They both need more computation than their less constrained counterparts 0-1000-2 and 0-10-2. We believe we know the reason for this. 500×500 and 6×6 are derived from Euclidean base problems. When the "Absolute Error" is minimized, it is likely that "physically close" vertices receive identical labels by the abstractions as their proximity to other vertices would carry the least absolute error. On the other hand, in the random problems there is no such notion of distance and it is likely that some relatively large cost edges are merged with small cost edges in the abstraction. Therefore "Absolute Error"can be misleading and applying penalties to keep precedences would be more appropriate. This conjecture is also supported by the b_0 values. In the case of 500×500-3 the lower bound for the start state is 72% of the optimal cost while it is only 40% for 0-1000-3 according to the smallest databases. When experimenting with different penalty values applied when a constraint

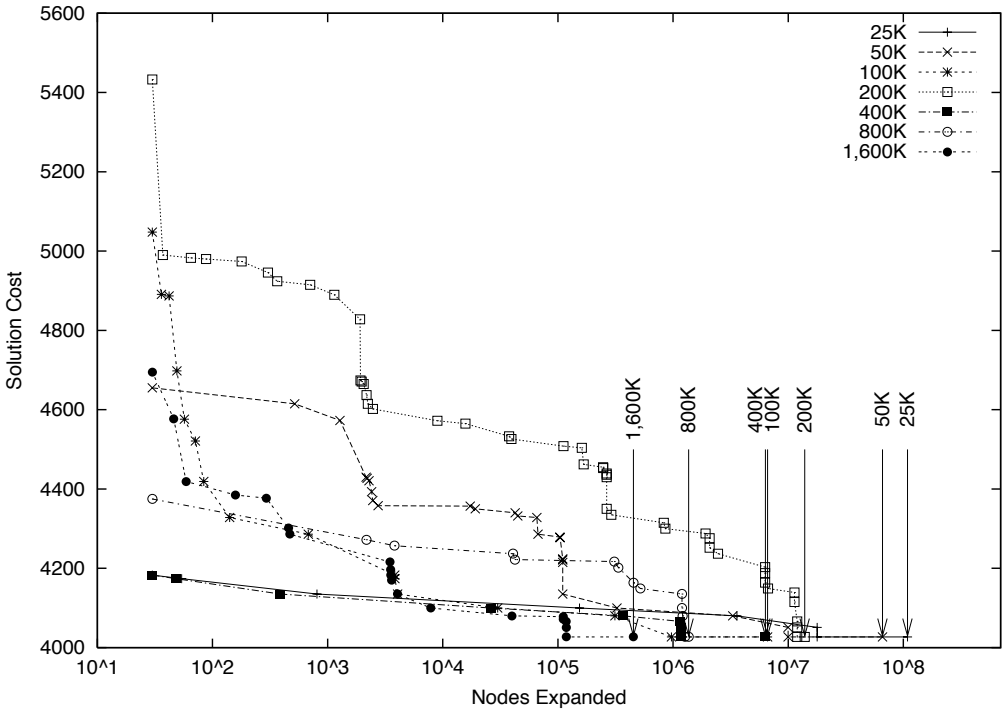

Figure 8: Solution Cost vs. Nodes Expanded for 500×500-3

is eliminated, we were able to bump up 0-1000-3's b_0 to 81% of the optimal cost with the same size pattern database. Our results also show that having larger databases pays off. In the case of 500×500-1 allocating a 5 times larger database results in a 148 fold reduction in nodes expanded. Increasing this size yet another 5 times and there is another a 21-fold reduction in nodes expanded. While these very high ratios are not typical, our experiments indicate that in general the trade-off between memory and search speed favors adding more memory.

We also performed experiments with recursive branch&bound and using transposition tables. For this experiment we used the smaller databases. In all cases we set a tolerance limit of 300 times the width of the pattern database (approximately 12 times its actual size). If this limit is reached, a new pattern database is built for the sole purpose of solving the largest current subtree excluding the start state. The node counts reported in Table 6 for the recursive branch&bound experiments (column marked Rec) include the nodes expanded in the subproblems as well. Because there is additional overhead of building more pattern databases, we also report the actual CPU times of the total runs. The column Rec+Trans lists the results of using recursive branch&bound and two levels of transposition tables. The results suggest that in most cases recursive branch&bound outperforms plain branch&bound even with additional overhead. In some cases the improvements are negligible or actually increase search effort. In other cases the improvements are significant. For example in the case of 500×500-1, the improvements are 7-fold and using transposition tables as well are almost 9-fold. For 0-1000-3 and 0-10-3 the improvements result in less search than using the 5-times larger pattern database (bold entries).

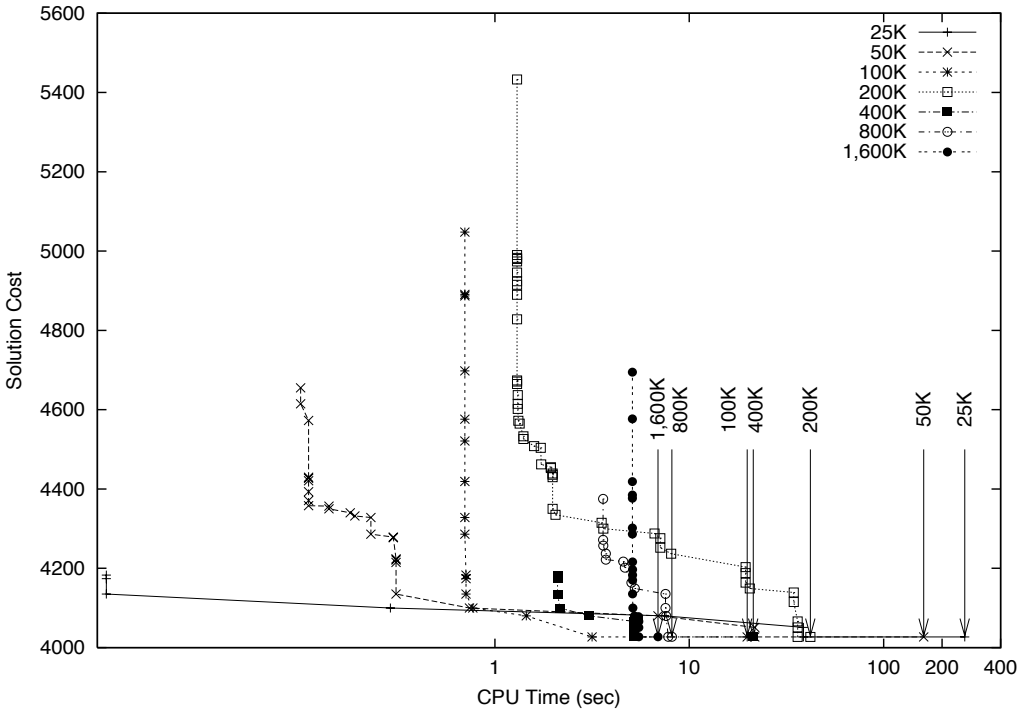

Figure 9: Solution Cost vs. CPU Time

7 Other Applications

Our lower bounds are derived from state space abstractions. These are induced by domain abstractions which are simple syntactic patterns that group states. While we tested our technique with the classic SOP only, we believe it could also be effective to solve other constrained TSPs as well.

Here, we detail two variations of the SOP for which our technique could generate lower bounds. We do not have experimental results at this time to actually evaluate their effectiveness.

Disjunctive Precedence Constraints

In the classical definition, the precedence is given explicitly. On the other hand one can imagine a more elaborate constraint structure where the constraint involves a choice. For example, "*one-of (3 and 4) must precede vertex 5*" or "*two-of (2, 3 and 4) must precede vertex 5*". Once we add disjunction we can express arbitrary boolean constraints of precedences. The precedences must be in the form

(*boolean constraint of vertices*) must-precede vertex

At this point, we would not allow a disjunctive constraint on the right-hand side of must-precede.

It is easy to see that we can encode such constraints into the successor generator function. Let s be a state representing the partial tour ending in the vertex with label v_c

$$s = (v_1, v_2, ..., v_k)[v_c]$$

When we create the successors, we simply check that the precedence constraints are observed for

SOP		50,000		250,000		1,250,000	
		1,000 Nodes	Time (sec)	1,000 Nodes	Time (sec)	1,000 Nodes	Time (sec)
500×500	1	3,046,341	21,790	20,596	208.29	964	10.57
	2	212,656	1,132	8,761	52.63	692	5.27
	3	66,025	278.13	7,750	37.92	460	3.07
	4	27,796	127.25	9,288	49.80	488	3.17
6×6	1	9,022,057	66,451	1,775,135	16,058	141,987	1,570
	2	1,154,906	5,818	35,283	206.61	3,476	25.25
	3	26,579	133.14	9,509	57.88	814	5.63
	4	988	4.84	176	1.06	4	0.03
0-1000	1	360,391	2,773	34,060	336.05	17,091	184.29
	2	25,661	152.81	3,787	26.34	159	1.51
	3	266,526	1,243	70,435	376.39	8,936	60.89
	4	176	1.04	32	0.22	4	0.04
0-10	1	118,071	721.58	33,489	248.34	3,006	32.74
	2	468	3.06	202	1.48	54	0.52
	3	59,310	290.84	22,045	126.68	10,943	61.89
	4	7,788	40.01	895	5.33	43	0.32

Table 5: Nodes Expanded and CPU Times of the Problem Set

v_c which is the right-hand side of the must-precede operator. In the abstract state, the domain abstractions must also be applied to the constraint definitions. For example, consider the constraint

two-of (2, 3 and 4) must-precede 5

and the domain abstractions

$$\phi_1(v) = \begin{cases} 2, & \text{if } v \in \{2,3\} \\ v, & \text{otherwise} \end{cases}$$

$$\phi_2(v) = \begin{cases} 2, & \text{if } v \in \{2,3,5\} \\ v, & \text{otherwise} \end{cases}$$

Under ϕ_1 the constraint reads

two-of (2, 2 and 4) must-precede 5

and under ϕ_2

two-of (2, 2 and 4) must-precede 2

Since domain abstractions do not eliminate vertices just relabel them, checking the abstract constraints is still trivial from state representation. With ϕ_1, we check if the tour has used up two 2's or a pair 2-4 labels. Otherwise the state is not generated. With ϕ_2, if the successor's current vertex is 2, we still have to make sure that either two other 2's or a pair of 2-4 are already part of the tour.

SOP		50,000					
		Plain		Rec		Rec + Trans	
		K Nodes	Time	K Nodes	Time	K Nodes	Time
500×500	1	3,046,341	21,790	405,420	3,220	350,431	2,817
	2	212,656	1,132	351,017	2,113	315,387	1,922
	3	66,025	278.13	33,327	146.79	26,871	118.18
	4	27,796	127.25	56,153	248.63	44,182	198.21
6×6	1	9,022,057	66,451	18,950,825	134,032	12,250,761	101,120
	2	1,154,906	5,818	30,4600	1,750	26,8862	1,551
	3	26,579	133.14	26,579	135.48	21,293	108.86
	4	988	4.84	988	4.96	988	4.98
0-1000	1	360,391	2,773	347,409	2,732	317,975	2,506
	2	25,661	152.81	17,865	112.49	17,686	109.62
	3	266,526	1,243	**51,707**	**230.21**	**48,021**	**212.11**
	4	176	1.04	176	1.03	176	0.03
0-10	1	118,071	721.58	61,936	420.93	61,798	423.42
	2	468	3.06	468	3.11	468	3.13
	3	59,310	290.84	**22,042**	**111.54**	**20,171**	**101.83**
	4	7,788	40.01	7,788	40.95	7,786	40.93

Table 6: Nodes Expanded and CPU Times of the Problem Set

Soft Precedence Constraints

We can also imagine that some precedences are cast in stone while others are allowed as far as some penalty is paid. It can be easily detected at the time the successors are generated if a penalty applies. The value of the penalty is added to cost of the partial tour corresponding to the successor. This is also true in the abstract space. However, we will only be able to add the penalty if the uniqueness of the penalized edge is preserved. That is, when the labels of the source and end vertices are not abstracted. Otherwise only the minimum of the original penalties between the preimage vertices can be applied. However a well designed error measure could optimize to preserve penalties in the abstract space.

8 Conclusion

We have defined a new way of deriving lower bounds automatically for the SOP. The lower bounds correspond to optimal tour completion costs in an abstraction of the original state space. These lower bounds are calculated at once and stored in a look-up table called the pattern database. We have introduced a novel way of building pattern databases which use different abstractions exploiting the level structure of the SOP state space lattice. The lower bound corresponding to the start state is a lower bound for all feasible solutions of the SOP. With our technique, we were able to tighten lower bounds for some instances of the unsolved `TSPLIB` problems.

We also have introduced many variations of depth-first branch&bound. Recursive branch&bound takes advantage of the property of the SOP that every consecutive sequence of vertices in a feasible tour is a SOP itself. This SOP is much smaller and our pattern database technique can be used to solve this subproblem independently. Bidirectional branch&bound exploits the fact that the SOP state space can be searched in both directions.

We experimentally confirmed that the larger the pattern database the faster the search. In fact, often the ratio of improvement in search speed far exceeds the ratio of memory increase. Unfortunately we could not establish a similar relationship between the size of the pattern database and the cost of the initial sequence of solutions. To obtain better solutions early, we also used a 3-opt heuristic tour improvement on each new solution we found. This, together with bi-directional search resulted in a solver which matched best known upper bounds in reasonable computational time for many of the unsolved `TSPLIB` instances. Using recursive branch&bound, we were able to prove a previously unknown optimal value for one instance from the `TSPLIB`.

The pattern database technique is a general strategy to derive lower bounds for state space search. We believe that it can also be used with other variations of constrained ATSPs and possibly to solve other scheduling/optimization problems. We also believe that it can be integrated with other solvers as well. For example, instead of a simple 3-opt, our algorithm could be seeded with the cheap solutions of the HAS-SOP solver of Gambardella *et al.* [10]. This way a good quality solution is guaranteed early. We also believe that the shortcomings of our pattern databases in finding solutions early could be solved by Ascheuer's [2] branch&cut. The solution to the LP relaxation is a fraction in $[0, 1]$ corresponding to each edge of the SOP. We could interpret these values as probabilities that tell how likely it is that the particular edge is included in the tour. We would sort branches by these probability values but still use our lower bounds for pruning.

9 Acknowledgement

I would like to thank my supervisor, Dr. Robert C. Holte, for his support, advice and encouragement during the completion of this work. I was also financially supported by an NSERC grant.

References

[1] N. Ascheuer. Hamiltonian path problems in the on-line optimization and scheduling of flexible manifacturing systems. PhD thesis, Technical University of Berlin, 1995.

[2] N. Ascheuer, M. Jünger, and G. Reinelt. A branch & cut algorithm for the asymmetric traveling salesman problem with precedence constraints. Computational Optimization and Applications, 17(1):61–84, 2000.

[3] N. Christofides, A. Mingozzi, and P. Toth. State space relaxation procedures for the computation of bounds to routing problems. In Networks, volume 11, pages 145–164, 1981.

[4] J. C. Culberson and J. Schaeffer. Searching with pattern databases. In Proceedings of the Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence, volume 1081, pages 402–416, 1996.

[5] E. Dijkstra. A note on two problems in connexion with graphs. In Numerische Mathematik, volume 1, pages 269–271, 1959.

[6] Stefan Edelkamp. Planning with pattern databases. Technical report, Institut für Informatik, Universität Freiburg, 2000.

[7] L. F. Escudero, M. Guignard, and K. Malik. A Lagrangian relax-and-cut approach for the sequential ordering problem with precedence constraints. In Annals of Operations Research, volume 50, pages 219–237, 1994.

[8] Fischetti. Facets of the asymmetric traveling salesman problem. MOR: Mathematics of Operations Research, 16, 1991.

[9] L. M. Gambardella and M. Dorigo. HAS-SOP: Hybrid ant system for the sequential ordering problem. Technical Report IDSIA-11-97, IDSIA, 1, 1997.

[10] L. M. Gambardella and M. Dorigo. An ant colony system hybridized with a new local search for the sequential ordering problem. INFORMS Journal on Computing, 12(3):237–255, 2000.

[11] Hermann Kaindl, Gerhard Kainz, Angelika Leeb, and Harald Smetana. How to use limited memory in heuristic search. In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, pages 236–242, 1995.

[12] R. Korf. Finding optimal solutions to Rubik's cube using pattern databases. In Proceedings of the Workshop on Computer Games (W31) at IJCAI-97, pages 21–26, 1997.

[13] A. Mingozzi, L. Bianco, and S. Ricciardelli. Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints. Operations Research, 45:365–377, 1997.

[14] W. Pulleyblank and M. Timlin. Precedence constrained routing and helicopter scheduling: Heuristic design. Technical Report RC17154, IBM, 1991.

[15] Dong-Il Seo and Byung-Ro Moon. A hybrid genetic algorithm based on complete graph representation for the sequential ordering problem. In Genetic and Evolutionary Computation – GECCO-2003, volume 2723 of LNCS, pages 669–680. Springer-Verlag, 2003.

[16] TSPLIB. http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

10 Appendix

10.1 Perfect Minimal Hash Function

I_l is a perfect minimal hash function over the states of $S'_l = \phi_l(S_l)$, not taking precedences into account. This means that given a state on $s'_l \in S'_l$, $I_l(s'_l)$ returns a unique integer in the interval $[0, \|S'_l\| - 1]$, where $\|S'_l\|$ is the total number of states on S'_l including those which represent partial tours that violate precedence constraints.

Recall that s'_l is represented as a set of abstract labels corresponding to the vertices not in the partial tour and the label of the last vertex in the partial tour. The number of labels available in s'_l for an n vertex SOP is $n - l$. This includes the label $k - 1$ for the terminal vertex of the tour. Therefore we have $n - l - 1$ "wild card" labels.

$$s'_l = (\underbrace{\text{available labels}}_{n-l-1}, k-1)[v_c]$$

To model this problem, imagine that we have x boxes, one box corresponding to each of the k abstract labels excluding the labels 0 and $k-1$ which are unique and reserved for the start and terminal vertices. $x = k-2$. The box i, for $1 \leq i \leq x$, contains $|\phi_l^{-1}(i)|$ pebbles with the same colour. No two boxes have pebbles with identical colours. There is at least 1 pebble in each box (we do not have redundant labels). The labels of the abstract state correspond to the pebbles we pick from the boxes. In particular, we have to choose 1 pebble for the current vertex (v_c) and then another $n-l-1$ pebbles from the remaining ones to represent the available vertex labels. To count the abstract states, we break down the problem into the these subproblems.

- choose 1 pebble from the $x = k-2$ boxes
- choose $n-l-1$ pebbles from the remaining ones

Subproblem 1 is clear. There are $k-2$ choices. For subproblem 2, we define a recurrence relation. Fix the order of the x boxes and number them with 1, 2, ..., x. Any order is good, but it has to be fixed ahead of time.

Let $f(a,b)$ be the number of ways one can select b pebbles from the boxes with numbers a, $a+1$, ..., x where each box contains n_a many identically coloured pebbles. The recurrence is

$$f(a,b) = \begin{cases} 0, & \text{if } b > \text{number of pebbles in the } a \text{ boxes} \\ 1, & \text{if } b = \text{number of pebbles in the } a \text{ boxes} \\ 1, & \text{if } b = 0 \\ 1, & \text{if } a = 1 \text{ and } b \leq n_a \\ \sum_{i=0}^{b} f(a-1, b-i) & \text{otherwise} \end{cases}$$

We can solve this recurrence with dynamic programming. The matrix of state variables T for $a = 4$ and $b = 4$ with $n_1 = 3$, $n_2 = 2$, $n_3 = 5$ and $n_4 = 2$ is

$T =$

a \ b	0	1	2	3	4
1	1	4	10	18	26
2	1	3	6	8	9
3	1	2	3	3	3
4	1	1	1	0	0

The value $T[1][4] = 26$ is the total number of ways the 4 pebbles can be picked from the 4 boxes.

Suppose we have chosen m_a pebbles from box a, $\sum m_a = b$. The value

$$h(.) = \sum_{a=1}^{x-1} \left(\sum_{i=0}^{m_a - 1} T\left[a+1\right] \left[\left(\sum_{j=a}^{x} m_j \right) - i \right] \right)$$

is unique for each choice of b pebbles in the interval $[0, T[1][b]]$.

Suppose we chose 2 pebbles from box 1, one pebble from box 3 and one pebble from box 4. Then the h value corresponding to this choice is

$$(T[2][4] + T[2][3]) + 0 + (T[3][2]) = 9 + 8 + 1 = 18$$

We need $k-2$ such dynamic programming tables, one for each choice of v_c. Let us order them according to the label values of of v_c. The hash value is obtained by calculating $h(.)$ and offsetting this value by the maximum entries of the tables that precede the one labeled v_c.

10.2 Checking Precedence Violations in Abstract States

Our domain abstractions reduce the dimension of the cost matrices between levels of the abstract space. This could inevitably lead to eliminated constraints and therefore result in the generation of abstract states that have no preimages in the original state space.

In this section, we present a technique to efficiently check two cases of such violations. For the discussions below, we consider only three vertex labels: a, b and c. Suppose there is a constraint that b must precede c in a tour. Now let $\phi(c) = a$. According to our merging rules, we have to eliminate this precedence because it is not true anymore that b must precede all a's and the invalid subpath aab occurs in abstract tours. We shall refer to this case as the aab violation. Now suppose there is a constraint that a must precede b and $\phi(a) = c$. For analogous reasons, the constraint is eliminated and the sequence bcc appears in tours in the abstract space. We claim that aab and bcc violations can be checked efficiently. Let A be the bag of vertex labels in s'_l which are part of the corresponding partial completion excluding the current vertex label b and let C be the bag of vertex labels which are still available. According to our representation $s'_l = (C)[b]$ from which A is also clear. The abstraction that generated s'_l is ϕ_l and C is the *original* cost matrix. We build two tables T_A and T_C. For each abstract label a, b, the entry $T_A[a][b]$ is the maximum number of non -1 entries amongst the columns of $C[\phi_l^{-1}(a)][\phi_l^{-1}(b)]$. To avoid aab violations, the number of a's in A must be at least $T_A[a][b]$. Similarly for each abstract label b, c, $T_C[c][b]$ is the maximum count of non -1 entries between the columns of $C[\phi_l^{-1}(c)][\phi_l^{-1}(b)]$. To avoid bcc violations, the number of c's in C must be at least $T_C[c][b]$.