Call-Based Dynamic Programming for the Precedence Constrained Line Traveling Salesman

Thierry Benoist[1], Antoine Jeanjean[2], and Vincent Jost[3]

[1] Innovation 24 - LocalSolver, Paris, France
tbenoist@localsolver.com
[2] Recommerce Solutions, Paris France
antoine.jeanjean@recommerce.com
[3] Grenoble-INP / UJF-Grenoble 1 / CNRS, G-SCOP UMR5272 Grenoble, France
Vincent.Jost@grenoble-inp.fr

Abstract. The Precedence Constrained Line Traveling Salesman is a variant of the Traveling Salesman Problem, where the cities to be visited lie on a line, the distance between two cities is the absolute difference between their abscissae and a partial ordering is given on the set of cities. Such a problem is encountered on linear construction schemes for instance. Using key dominance properties and lower bounds, we design a call-based dynamic program able to solve instances with up to 450 cities.

1 Introduction

The Line-TSP is a variant of the Traveling Salesman Problem (TSP) where the cities to be visited lie on a line, and the distance between two cities is the absolute difference between their abscissae. Although trivial in this pure formulation, the problem becomes interesting when side-constraints are added. For instance, [13] considers the case where each city must be visited within a certain time-window. The present paper deals with the case where a partial ordering is given on the set of cities that is to say that some precedence constraints *A must be visited before B* must be satisfied. In practice this problem is encountered on linear construction schemes when a set of partially ordered tasks (up to several hundreds) must be performed by a resource whose traveling distance must be minimized, like an excavation engine on a highway construction site for instance [8]. To the best of our knowledge, this problem was not studied in the literature before, but its NP-completeness was established as a special case in [3]. Linear structures also occur in *N-line TSP* [4,12] namely an Euclidian TSP with a limited number of of different abscissae and also in the so-called *convex hull and line TSP* [5,6].

After a formal definition of the problem we introduce a key dominance property.In section 2 we define a lower bound based on the splitting of the line in sections. Finally, we propose a call-based dynamic programming approach, where branches are pruned with our lower bound. This algorithm is experimented in section 4.

1.1 Problem Definition and Notations

Definition 1. *The Precedence Constrained Line TSP (PC-Line-TSP) reads as follows.*

Given a set P of n cities, each with an abscissa X_i ($i \in [1, n]$), a partial order $\prec$ on P and an integer K; find a permutation V of P such that :

$$\begin{array}{l} \sum_{i=1}^{n} | X_{V(i)} - X_{V(i-1)} | \leq K, \textit{ with } X_{V(0)} = 0 \\ \forall (i, j) \in P^2 \textit{ such that } i \prec j, \; V^{-1}(i) < V^{-1}(j), \\ \textit{with } V^{-1}(i) \textit{ the position of city } i \textit{ in the permuation.} \end{array} \tag{1}$$

The above sum of differences of abscissae will be referred to as the *length* of the permutation. We will denote by m the size of $\prec$ that is to say the number of ordered pairs defining this partial order. Figure 1 represents a instance of PC-Line-TSP with four cities, the Hasse diagram being represented on the left side. The dotted arrows represent a feasible solution (visiting order).

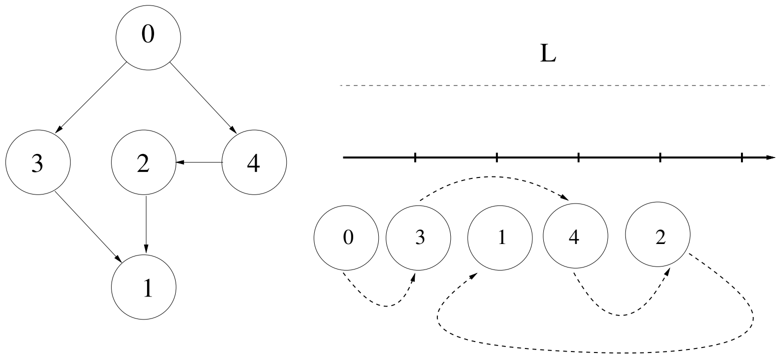

Fig. 1. A Precedence Constrained Line TSP

1.2 Properties

Definition 2. *In a solution, a* procrastination *is a city which is passed through without being visited whereas all its predecessor have already been visited.*

Definition 3. *A solution is called* dominating *(ou* non-procrastinating*) is it contains no procrastination. In other words it never passes through an abscissa without visiting all available cities at this abscissa (namely cities whose predecessors have already been visited).*

Lemma 1. *There exists a* dominating *optimal solution for each problem.*

Proof. Let V be a non dominating permutation, that is to say that there exists at least one procrastination city d. By definition there are two consecutive cities a and b in the permutation such that:

$$X_d \in [X_a, X_b[\quad or \quad X_d \in]X_b, X_a] \tag{2}$$

and such that all predecessors of d (possibly including a) are before a in permutation V (see Figure 2), while city d is visited later, between two cities c and e. Note that c may equal b and that e may not exist (in which case d is the last city of the permutation), without affecting the validity of the reasoning below.

Let V' be the permutation obtained from V by moving city d between a and b. V' satisfies the partial order because all predecessors of d are before a in both permutations. Now we prove that the length or permutation V' is smaller or equal to the length or permutation V.

- the path $a \rightarrow d \rightarrow b$ is equal to path $a \rightarrow b$ (d being between a and b).
- the path $c \rightarrow e$ is smaller or equal to path $c \rightarrow d \rightarrow e$ since distances on a line satisfy the triangular inequality (and if e does not exists, the path $c \rightarrow d$ is merely removed.

Repeating this transformation, all procrastinations can be eliminated, while preserving or decreasing the length of the permutation, thus building a dominating permutation of equal or smaller length. Hence any problem has an optimal dominating solution. □

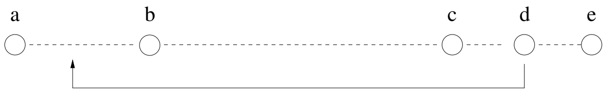

Fig. 2. Procrastination removal

Corollary 1. *A solution can be written as a sequence of t abscissae with $t \leq n$. This sequence (or* path*) induces a visiting order (unique but for the visiting order of cities at the same abscissa). Hence the PC-Line-TSP can be reformulated as a question: is there a sequence of abscissae (or path) whose length is smaller than K and which visits all cities ?*

Corollary 2. *The special case where cities can take only two different abscissae ($|\{X_i, i \in [0, n]\}| = 2$) is polynomial. Its optimal path alternates the two abscissae. It can be computed in linear time.*

1.3 Complexity

We have seen above that the problem is polynomial when limited to 2 abscissae. In the general case this problem was proven to be NP-hard by [3], by reduction from the Shortest Common Supersequence problem. It is NP-hard as soon as the number of number of abscissae if larger or equal to 3 (see [9]).

2 Lower Bounds

In this section we compute lower bounds for the PC-Line-TSP problem.

Trivial bound LB_0. The distance between the maximum and minium abscissae plus the distance from 0 the closest extremity is a lower bound of the traveled distance.

$$R = \max_{i \in [0,n]} X_i$$
$$L = \min_{i \in [0,n]} X_i$$
$$LB_0 = R - L + \min(|R|, |L|)$$

Definition 4. *A* section $[X_l;\ X_r]$ *is a pair of consecutive abscissae. The linear line is thus made of less than n disjoint sections (see Figure 4). The length of section* $[X_l;\ X_r]$ *is* $X_r - X_l$*. In what follows abscissae are indexed (from left to right) from 0 to p, with* $p \leq n$ *. The section between abscissae* $k-1$ *and* k *is referred to as the* k^{th} *section, and its with is denoted* w_k*.*

In this section we define a lower bound LB_1 based on this decomposition into sections and on the polynomial 2-abscissae case evoked in corollary 2.

For each section k defined by abscissae $[X_l;\ X_r]$ we build a problem with two abscissae, setting to zero all abscissae smaller or equal to X_l and setting to $w_k = X_r - X_l$ all abscissae larger or equal to X_r. The algorithm presented below computes a lower bound to the number of times this section will be crossed.

Algorithm. The Algorithm 1 details the computation of this lower bound. The initial scan of the graph modifying abscissae is done in $O(n)$. We define $opt_l(k)$ (resp. $opt_r(k)$), the minimum number of times section k will be crossed (see Figure 3) considering that the terminal city (the last city in the permutation) lies to the left (resp. to the right) of the section.

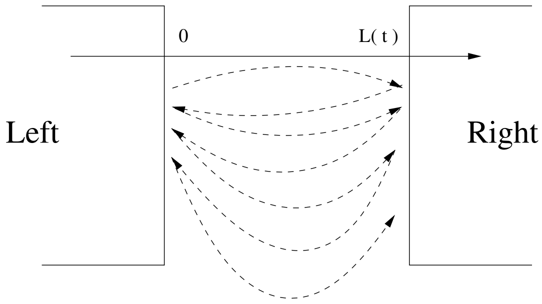

Fig. 3. Scanning a section

We denote by $B(k)$ the lower bound of the total length when the last abscissa is abscissa k ($k \in [0, p]$). It is defined by the following recursive formula:

$$\begin{array}{l} B(0) = \sum_{k \in [1,\ p]} opt_l(k) \\ \forall\ k \in [1,\ p],\ B(k) = B(k-1) - opt_l(k-1) + opt_r(k-1) \end{array} \quad (3)$$

Let F be the set of abscissae having at least one city without successor in the partial order. Since the final city of any permutation will belong to F, the following expression is a lower bound of the traveled distance of the PC-Line-TSP.

$$LB_1 \;=\; \min_{k \,\in\, F} B(k) \tag{4}$$

For each section $k \in [1,p]$, $opt_l(k)$ is computed in $O(\ n\ +\ m\ )$ with n the number of cities and m the size of the partial order. $B(\mathit{0})$ is computed in $O(\ p\ )$ with $p \ \leq\ n$ and each $B(k)$ is computed in $O(1)$. Finally the complexity of the computation of this lower bound is $O(\ p\ (\ n\ +\ m\ )\ )$. Since p is smaller than n et m is smaller than n^2, the complexity is cubic in the worst case.

Algorithm 1. Bound-by-section

```
input  : The set of abscissae, cities and the partial order
output: A lower bound of the total traveled distance
begin
    for $k \in [1,p]$ do
        Define the two-abscissae problem associated to section $k$, with starting
        city at abscissa 0
        Distance = 0
        CurrentAbscissa = 0
        Visit all available cities at CurrentAbscissa
        while Some cities remains to be visited do
            Change CurrentAbscissa to the other abscissa
            $Distance \mathrel{+}= w_k$
            Visit all available cities at CurrentAbscissa
        if $CurrentAbscissa = w_k$ then
            $opt_l(t) = Distance + w_k$ $opt_r(t) = Distance$
        else
            $opt_l(t) = Distance$ $opt_r(t) = Distance + w_k$
    $B(\mathit{0}) = \sum_{k\in[1,p]} opt_l(k)$
    for $k \in [1,\ p]$ do
        $B(k) = B(k-\mathit{1}) - opt_l(k-1) + opt_r(k-1)$
    return $\min_{k\in F} B(k)$
```

Example. Consider a PC-Line-TSP made of six cities partially ordered as in Figure 4. Here the set of possible terminal cities (cities without successors) is $\{N_{\mathit{1}},\ N_{\mathit{4}},\ N_{\mathit{5}}\}$. The algorithm 1 considers each of the five sections one by one. On the left of Figure 4, section 2 (between abscissae 2 and 5) is emphasized, and the two-abscissae problem associated to this section is illustrated (defined on abscissae 0 and $w_{\mathit{3}} \ =\ 3$). The computation of $opt_l(2)$ and $opt_r(2)$ starts with the visit of city N_0 on the left side. N_1 cannot be visited yet due to its two unvisited predecessors N_2 and N_3. A first crossing of the section is needed then.

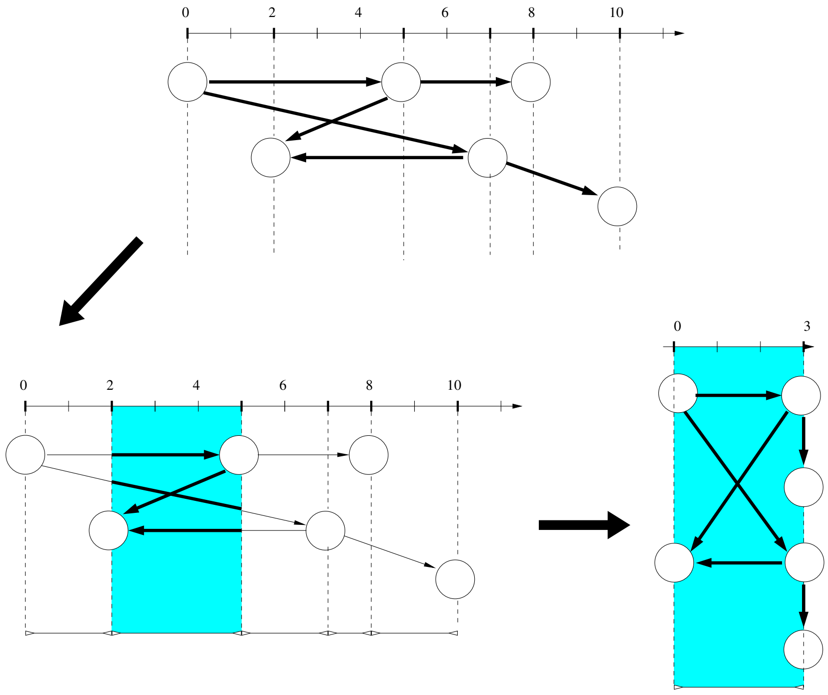

Fig. 4. A simple PC-Line-TSP instance and the two-abscissae problem attached to is second section

On the right side, N_2 and N_3 are visited, thus allowing the visit of N_4 et N_5. The section need to be crossed a second time in order to visit N_1 on the left side, thus completing the path. Finally $B(2) \ = \ 2 \ \times \ 3 \ = 6$.

For each section $k \in [1, 5]$, $opt_l(k)$ et $opt_r(k)$ take the following values :

$$\begin{array}{lllllllllllllllll} opt_l(1) & = & 2 & \times & 2 & = & 4 & et & opt_r(1) & = & 1 & \times & 2 & = & 2 \\ opt_l(2) & = & 2 & \times & 3 & = & 6 & et & opt_r(2) & = & 3 & \times & 3 & = & 9 \\ opt_l(3) & = & 2 & \times & 2 & = & 4 & et & opt_r(3) & = & 3 & \times & 2 & = & 6 \\ opt_l(4) & = & 2 & \times & 1 & = & 2 & et & opt_r(4) & = & 1 & \times & 1 & = & 1 \\ opt_l(5) & = & 2 & \times & 2 & = & 4 & et & opt_r(5) & = & 1 & \times & 2 & = & 2 \end{array} \tag{5}$$

and we can compute the following $B(k)$:

$$\begin{array}{lllllllllllllll} B(0) & = & \sum_{k \in [1,p]} & opt_l(k) & = & 4 & + & 6 & + & 4 & + & 2 & + & 4 & = & 20 \\ B(1) & = & B(0) & + & opt_l(0) & + & opt_l(0) & = & 20 & - & 4 & + & 2 & = & 18 \\ B(2) & = & B(1) & + & opt_l(1) & + & opt_l(1) & = & 18 & - & 6 & + & 9 & = & 21 \\ B(3) & = & B(2) & + & opt_l(2) & + & opt_l(2) & = & 21 & - & 4 & + & 6 & = & 23 \\ B(4) & = & B(3) & + & opt_l(3) & + & opt_l(3) & = & 23 & - & 2 & + & 1 & = & 22 \\ B(5) & = & B(4) & + & opt_l(4) & + & opt_l(4) & = & 22 & - & 4 & + & 2 & = & 20 \end{array} \tag{6}$$

Since $F \ = \ \{N_1, \ N_4, \ N_5\}$, the final bound is $min(B(1), \ B(4), \ B(5)) \ = \ 18$.

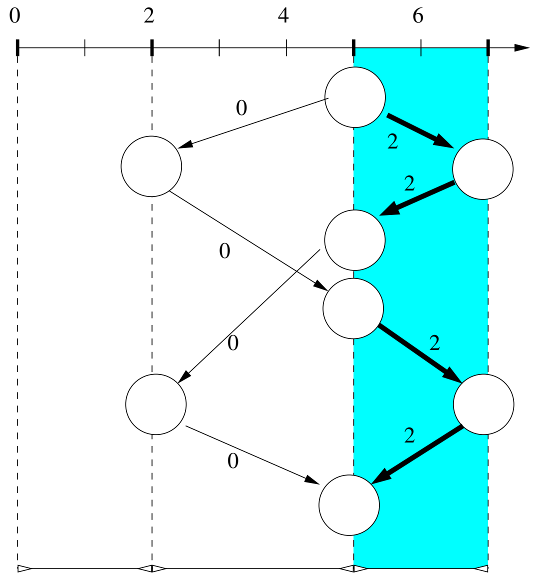

Fig. 5. Suboptimal lower bound

Counter-Example. The example on Figure 5 illustrates the non-optimality of lower bounds LB_1. There is only one possible terminal city (N_7). On section $[2, 5]$, we obtain a lower bound equal to 6 and on section $[5, 7]$ we obtain a lower bound equal to 4, hence $LB_1 = 10$. However, the optimal solution is 20 : $N_0\ -2 \to\ N_2\ -2 \to\ N_3\ -3 \to\ N_1\ -3 \to\ N_4\ -2 \to\ N_6\ -2 \to\ N_7\ -3 \to\ N_5\ -3 \to\ N_7$:

3 Exact Algorithm

3.1 Dynamic Programming

Any dominating solution to the PC-Line-TSP problem can be expressed as a sequence of left/right decisions. That is to say that each time the current abscissa has no available city, we are facing a binary choice: either go to the closest abscissa with available cities to the left or to the closest abscissa with available cities to the right. It means that a brute force enumeration of all dominating solutions has complexity $O(2^n)$, while the number of possible permutations is $n!$. We present in this section a dynamic programming approach similar to the one proposed by [7] for the classical TSP, observing that at any moment in the search, the remaining distance to be traveled only depends on the current abscissa and on the set of remaining cities. The worst case complexity of this algorithm remains $O(2^n)$, but thanks to the non-procrastination rule and to the partial ordering of cities many sets of cities cannot be encountered as a set of remaining cities, what makes this algorithm very effective in practice.

In essence, the minimum length $L_{min}(x,\ Q)$ for visiting a set of cities $Q \subseteq P$ starting from abscissa x and subject to partial order $\prec$ can be expressed with the following recursive formula, where $A(y)$ denotes the set of available[1] cities at abscissa y while X_R (resp. X_L) is the closest abscissa to the right of x (resp. to the left of x) such that $A(X_R) \neq \varnothing$ (resp. $A(X_L) \neq \varnothing$). Note that X_R and X_L are functions of x and Q, but these parameters are omitted in the remaining of the paper for the sake of readability. Without loss of generality we assume $A(x) = \varnothing$.

$$\begin{aligned} L_{min}(x,\ \varnothing) &= 0 \\ L_{min}(x,\ Q) &= \min(\quad X_R - x + L_{min}(X_R,\ Q \setminus A(X_R)), \\ &\qquad\qquad x - X_L + L_{min}(X_L,\ Q \setminus A(X_L)) \end{aligned}$$

This dynamic programming approach yields the optimal solution of the PC-Line-TSP as $L_{min}(0,\ P)$. Inspired by [1] we design a call-based dynamic program based on this recursive formula. Following their terminology, call based dynamic programming consists in implementing a classical tree search where the optimum for each subtree is stored and re-used each time the same sub-tree is encountered (same x and same Q in our case). Compared to bottom-up dynamic programming implementation, the call-based approach allows introducing lower-bounds, upper-bounds and heuristics in order to speed up the search.

As detailed in algorithm 2, the central function becomes $L_{min}(\ x,\ Q,\ U)$ where U is an upper bound, and the optimum of the problem is $L_{min}(0,\ P, +\infty)$. As soon as a first solution is found, it is used to define upper-bounds for other branches thus excluding solutions leading to a total traveled distance larger or equal to the best found so far. The trivial lower-bound LB_0 defined in section 1 is used to eliminate such sub-optimal solutions as soon as possible. In section 4 we will also give the results obtained when using bound LB_1 instead.

The DP-labeled lines are specific to dynamic programming[2]: $storedBest[x, Q]$ is the minimum distance for visiting cities of Q when starting from abscissa x, hence before exploring a subtree (x, Q), the algorithm always check whether its minimum distance is already known (that is to say if $storedBest[x, Q]$ is defined). Similarly $storedLB[x, Q]$ is the best known lower-bound to this distance. Indeed once a subtree was vainly explored searching for a solution with a traveled distance strictly smaller than U, this information is worth storing because later in the search this subtree may be considered again with some upper bound U'. Then if $U' \leq U$ the re-exploration of this subtree is avoided. In theory the number nodes in the search tree can be larger when using lower bounds because the same sub-problem (x, Q) can be explored several time with increasing upper bounds. However the pruning effect largely compensates for this in practice. For

[1] These cities may be partially ordered by $\prec$ but they can all be visited if we reach abscissa y.

[2] In other words, removing the DP-labeled lines results in a classical tree search algorithm.

Algorithm 2. $L_{min}(x, Q, U)$

```
input  : Current abscissa $x$, remaining cities $Q$, an upper bound $U$
output: The length of the best solution if $< U$, $U$ otherwise
begin
       if $Q = \varnothing$ then Return 0
DP     if $storedBest[x, Q]$ then Return $storedBest[x, Q]$
DP     if not($storedLB[x, Q]$) then $storedLB[x, Q] = LB_0(x, Q)$
DP     if $storedLB[x, Q] \geq U$ then Return U
       if $LB_0(x, Q) \geq U$ then Return $U$
       $Best = U$
       if $X_R \neq +\infty$ then
           $D_{right} = X_R - x$
           $Best = D_{right} +$
                   $L_{min}(X_R, Q \setminus A(X_R), Best - D_{right})$
       if $X_L \neq -\infty$ then
           $D_{left} = x - X_L$
           $Best = D_{left} +$
                   $L_{min}(X_L, Q \setminus A(X_L), Best - D_{left})$
DP     if $Best < U$ then $storedBest[x, Q] = Best$
DP     else $storedLB[x, Q] = U$
       Return min($Best$, $U$)
```

instance once a solution of length 35 has been found, a certain state (x, Q) might be reached after a traveling distance of 10 (visiting cities in $P \setminus Q$) hence with an upper bound of 25; but later in the search this sate may be reached after a traveling distance of 9 that is to say with an upper bound of 26, in which case this subtree must be explored again.

With $storedBest[x, Q]$ and $storedLB[x, Q]$ stored in hashtables, these values can be accessed and updated in constant time. Provided that $A(y)$ is dynamically maintained for each abscissa y (in $O(m)$ amortized complexity), X_R and X_L can be obtained in $O(p)$ (recall that m is the size of the partial order and p is the number of different abscissae). Maintaining the leftmost and rightmost abscissae in Q makes sure that LB_0 is computed in constant time. If LB_1 is used instead of LB_0, its complexity is $O(\ p\ (\ n\ +\ m\ )\ )$ as shown in section 1.

3.2 Heuristics

In algorithm 2, the right side is systematically explored before the left side. However, different strategies can be applied. For instance a *NearestNeighbor* heuristic would consist in starting with the left side when $x - X_L < X_R - x$ (and starting with the right side otherwise). Recall that for the Euclidian TSP this simple heuristic averages less than 25% above the Held-Karp lower bound [10] and is guaranteed to remain within a $\frac{1}{2}(\log_2(N)+1)$ ratio of the optimal solution

[11]. Alternatively an *Inertial* heuristic would consist in continuing rightwards if and only if the current abscissa was reached from the left. Finally, based on our lower bound LB_0 or LB_1, an $A\star$ heuristic consists in evaluating the lower bound on each branch and then start with the most promising one, that is to say the one with the smallest lower bound.

3.3 Dominance Rules

Lemma 2. *Let Z be a subset of P which is totally ordered by $\prec$, and let T be the subset of all cities of $P \setminus Z$ necessarily visited by a path visiting Z. If T has no successor in P' ($\nexists a \in T, b \in P', a \prec b$), then the problem restricted to cities in $P' = P \setminus T$ has the same optimal value as the initial problem.*

Proof. Clearly any solution of the initial problem is also a solution of the problem limited to P'. Inversely any solution path of the problem limited to P' is a super-sequence of the the sequence of abscissae of set Z (ordered by $\prec$), hence the cities of T can be inserted in the solution without increasing its length, while respecting the precedences $a \prec b$ with $b \in T$. Precedences internal to P' are satisfied since these insertions do not modify the ordering of cities of P'. By hypothesis no precedence is defined from T to P'. Finally the obtained permutation has the same length as the initial solution and satisfies the partial order on P. $\square$

Corollary 3. *In particular, this dominance rule can be applied for any leftmost or rightmost city of P (any singleton being totally ordered by $\prec$). It means that any centrifugal connex part of the Hasse Diagram (that is made of precedences $a \prec b$ with b farther from 0 than a) can be removed from P without affecting the value of the optimal solution.*

Corollary 4. *PC-Line-TSP is fixed parameter tractable when parameterized by m.*

Proof. At least $n - m$ cities do not appear in the Hasse diagram, and thus can be removed from P without affecting the value of the optimal solution (provided that two extreme cities are kept). Consequently this reduced problem has a size limited to $m + 2$. $\square$

3.4 Examples

As mentioned above the worst case complexity of our dynamic programming algorithm is the same as the one of a complete scan (2^n). Before demonstrating the practical gains of dynamic programming in the next section, we exhibit below two structures for which the complexity of the dynamic program is much better.

In Figure 6, a complete scan has complexity $O(2^n)$ (one binary choice per layer), whereas dynamic programming has complexity $O(4n)$ (4 states per layer). In Figure 7 cities are distributed around a central starting city with no precedence between them, then the dynamic program explores n^2 states while complete scan remains exponential (2^n). In the latter case, the use of the dominance rule of corollary 3 reduces the number of states to 2 (all cities but two can be removed).

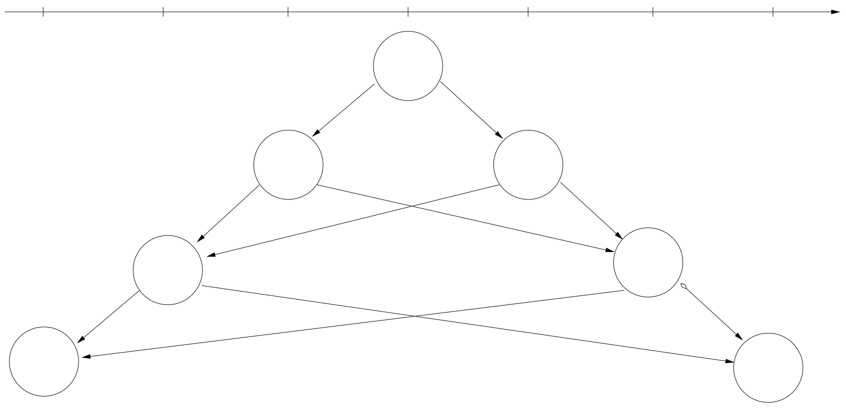

Fig. 6. Special partial orders(1)

4 Computational Results and Conclusion

4.1 Problem Instances

For generating an instance with n cities and k distinct abscissae, we start with building a partial order on [1,n]. For each pair (i, j) with $i < j$, a precedence $i \prec j$ is generated with probability p. Three remarks can be made on this partial order:

- It is a partial order because the generated directed graph contains no cycle (by construction all arcs are oriented toward growing integers).
- Several other Directed Acyclic Graph would represent the same partial order. We can define the density of the partial order as the number of arcs included in the transitive closure of this graph divided by the number of arcs in the total order $(n(n-1)/2)$. A total order has a density of 100%.
- this simple method does not ensure that the generated partial orders are uniformly distributed among the set of all possible partial order. However our goal here is not to extract statistical properties but merely to generate a set of instances for comparing our algorithms. For references on the uniform generation of partial orders see [14] and [2].

Once this partial order is generated, k distinct abscissae are randomly drawn in [0,100]. Cities receive random abscissae from this set.

We generated 44 random instances of the PC-Line-TSP, with a number of cities (cities) from 100 to 450, a density from 5% to 85% (instances with a density of 100% were discarded) and a number of distinct abscissae from 23 to 99. We also generated 20 instances with 100 to 450 cities on 3 distinct abscissae, the first NP-Complete value (see section 1.3): any of our algorithms could solve any of these 3-abscissae instances in less than one second.

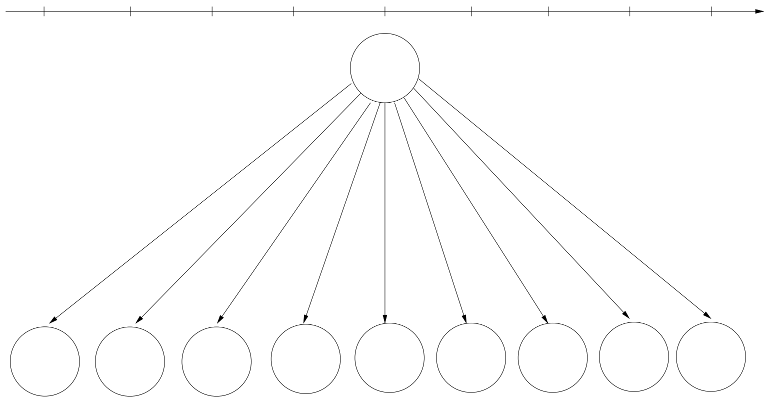

Fig. 7. Special partial orders (2)

4.2 Results

For each instance we compared 4 algorithms:

1. Our complete algorithm with bound LB_1 (bound by section defined in section 1), with our reduction algorithm enabled (corollary 3), and using heuristic *NearestNeighbor*.
2. The same algorithm as 1 using heuristic $A\star$
3. The same algorithm as 1 with bound LB_0
4. The same algorithm as 1 without our reduction algorithm

Table 1 summarizes average results obtains with these four algorithms with a time limit set to 600 seconds. Our reference algorithm found the optimal solution of 36 of the 44 instances (with both heuristics). Disabling the reduction algorithm leads to a score of 33/44 while using bound LB_0 instead of LB_1 we obtain only 27/44. In terms of completion time our second algorithm (with heuristic $A\star$) obtains the best results, with an average time of 113 seconds vs 129 seconds with heuristic *NearestNeighbor*. The impact of this heuristic is more significant when comparing the time to obtain the best solution (when both completed the search in the allocated time): with $A\star$, the time to obtain the best solution is divided by 3 on average: 31 seconds against 100 seconds.

Table 2 reports the results obtained by each algorithm on the 8 instances that could not be solved within the allocated 600 seconds. Our second algorithm (with $A\star$ heuristic) always obtains the best solution. Consequently all gaps in this table are given with respect to the results of this best algorithm. The rightmost column gives the value of our bound by sections LB_1 and the corresponding optimality gap.

As shown in table 3, the density of the partial order plays a important role in the hardness of an instance. Looking at results of algorithm 3 we see that the higher the density the easier the instance, with a number of proven optimum

Table 1. Average results on the 44 PC-Line-TSP instances (time limited to 600 seconds)

	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
Pruning Heuristic	LB_1 + reduction NearestNeighbor	LB_1 + reduction $A\star$	LB_0 + reduction NearestNeighbor	LB_1 NearestNeighbor
Number of proven optimum	36/44	36/44	27/44	33/44
Average time to complete	129s	113s	239s	153s
Average time to obtain best solution*	100s	31s	-	-

(*)on the 36 instances solved by 1 and 2

Table 2. Results on the 8 unsolved instances

	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4	Lower Bound
Pruning Heuristic	LB_1 + reduction NearestNeighbor	LB_1 + reduction $A\star$	LB_0 + reduction NearestNeighbor	LB_1 NearestNeighbor	LB_1
Pb200	**761**	**761**	**761**	**761**	619 (-19%)
Pb350	1904 (+3.0%)	**1848**	2036 (+10.2%)	2130 (+15.3%)	1563 (-15%)
Pb400A	4230 (+1.4%)	**4170**	4236 (+1.6%)	5118 (+22.2%)	3963 (-5%)
Pb400B	4500 (+0.1%)	**4498**	4500 (+0.1%)	4548 (+1.1%)	4268 (-5%)
Pb400C	3193 (+1.9%)	**3133**	3583 (+14.4%)	3889 (+24.1%)	2946 (-6%)
Pb400D	2345 (+5.0%)	**2233**	2477 (+10.9%)	2739 (+22.7%)	2171 (-3%)
Pb450A	3098 (+3.7%)	**2988**	3156 (+5.6%)	3688 (+23.4%)	2723 (-9%)
Pb450B	3154 (+20.3%)	**2876**	3460 (+9.7%)	3386 (+17.7%)	2653 (-8%)
Average gaps	+7.9%		+3.1%	+15.9%	-9%

increasing from 5 to 8. Indeed when the partial order is denser, the number of feasible permutations is smaller. The extreme case is when the density is 100% and only one permutation is allowed. Comparing algorithm 1 and 3 we see that using lower bound LB_1 pays off on all range of densities. As for the reduction algorithm its impact is higher on smaller densities, because sparse partial orders are more likely to contain centrifugal connex parts. Concerning the number of abscissae, we noticed that the number of distinct abscissae can increase the hardness of the problem or at least the complexity of our algorithms. The median number of abscissae is 60 is our benchmark. Our best algorithm (number 2) solved instances with less than 60 abscissae in 90 seconds on average (proving 21 optimum values out of 22) while instances with more than 60 abscissae are solved in 246 seconds on average (with 15 proven optimum out of 22).

Table 3. Impact of the density of the partial order (time to complete in seconds and number of proven optimum in parenthesis)

	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
Pruning Heuristic	LB_1 + reduction NearestNeighbor	LB_1 + reduction $A\star$	LB_0 + reduction NearestNeighbor	LB_1 NearestNeighbor
First quartile [5% to 20%]	154s (10/11)	127s (10/11)	368s (5/11)	259s (7/11)
Second quartile [20% to 48%]	245s (7/11)	198s (7/11)	344s (6/11)	277s (7/11)
Third quartile [48% to 73%]	186s (10/11)	180s (10/11)	331s (8/11)	197s (10/11)
Fourth quartile [73% to 85%]	164s (9/11)	162s (9/11)	347s (8/11)	161s (9/11)

4.3 Conclusion

Our call-based dynamic program, manages to solve to optimality instances with up to 450 cities. Examining the results we see that our lower bound LB_1, based on the splitting of the line by sections, makes possible the resolution of dense and large instances. On the other end, our domination rule dramatically improves performance on sparse instances. Finally, thanks to the call-based implementation, the algorithm can find good solutions, even for instances whose optimum could not be found within 600 seconds. In this context founding an $A\star$ heuristic on our lower bound allows finding better solutions faster.

References

1. de la Banda, M.G., Stuckey, P.J.: Dynamic programming to minimize the maximum number of open stacks. INFORMS Journal on Computing 19(4), 607–617 (2007)
2. Brightwell, G.: Models of random partial orders, pp. 53–84. Cambridge University Press (1993)
3. Charikar, M., Motwani, R., Raghavan, P., Silverstein, C.: Constrained tsp and low-power computing. In: Dehne, F., Rau-Chaplin, A., Sack, J.-R., Tamassia, R. (eds.) WADS 1997. LNCS, vol. 1272, pp. 104–115. Springer, Heidelberg (1997)
4. Cutler, M.: Efficient special case algorithms for the n-line planar traveling salesman problem. Networks 10(3), 183–195 (1980)
5. Deineko, V.G., van Dal, R., Rote, G.: The convex-hull-and-line traveling salesman problem: A solvable case. Information Processing Letters 51(3), 141–148 (1994)
6. Deineko, V.G., Woeginger, G.J.: The convex-hull-and-k-line travelling salesman problem. Inf. Process. Lett. 59(6), 295–301 (1996)
7. Held, M., Karp, R.M.: A dynamic programming approach to sequencing problems. In: Proceedings of the 1961 16th ACM National Meeting, pp. 71.201–71.204. ACM, New York (1961)
8. Jeanjean, A.: Resource scheduling optimization in mass transportation problems. In: 12th International Conference on Project Management and Scheduling, PMS 2010 (2010)
9. Jeanjean, A.: Recherche locale pour l'optimisation en variables mixtes: Méthodologie et applications industrielles. Ph.D. thesis, Laboratoire d'informatique de Polytechnique (2011)
10. Johnson, D.S., Mcgeoch, L.A.: The Traveling Salesman Problem: A Case Study in Local Optimization. John Wiley and Sons, Chichester (1997)
11. Rosenkrantz, D.J., Stearns, R.E., Lewis II, P.M.: An analysis of several heuristics for the traveling salesman problem. 6(3), 563–581 (1977)
12. Rote, G.: The n-line traveling salesman problem. Networks 22, 91–108 (1991)
13. Tsitsiklis, J.N.: Special cases of traveling salesman and repairman problems with time windows. Networks 22, 263–282 (1992)
14. Gehrlein, V.W.: On methods for generating random partial orders. Operations Research Letters 5(6), 285–291 (1986)