Branch-and-bound: an example

Giovanni Righini

Università degli Studi di Milano

Operations Research Complements

Università degli Studi
di Milano

The Linear Ordering Problem

The Linear Ordering Problem (LOP) is an $\mathcal{NP}$-hard combinatorial optimization problem that consists in finding an optimal permutation of a given set $\mathcal{N}$ of items.

The objective function value depends on whether item i precedes or follows item j for each (i, j) pair of items in $\mathcal{N}$.

The LOP has many applications, ranging from marketing to psychology, from scheduling to archaeology.
A polyhedral study of the LOP is due to Grötschel et al. (1984a, 1985a, 1985b).
More references to applications can be found in Reinelt (1985).

The LOP model

Given a complete digraph $\mathcal{D} = (\mathcal{N}, \mathcal{A})$, find an optimal spanning acyclic tournament in $\mathcal{D}$.

An acyclic tournament is an acyclic digraph $\mathcal{T} = (\mathcal{N}, \mathcal{A}')$ such that for every two distinct vertices i and j in $\mathcal{N}$ the arc set $\mathcal{A}'$ contains exactly one arc with endpoints i and j.

Acyclic tournaments are in one-to-one correspondence with permutations of the vertices: $\Pi : \{1, \ldots, N\} \mapsto \mathcal{N}$, where $N = |\mathcal{N}|$.

$\Pi(k)$ indicates the vertex in position k.
$\Pi^{-1}(i)$ indicates the position in which vertex i is.

The LOP-CC

The Linear Ordering Problem with Cumulative Costs (LOP-CC) arises from an application in the sector of UMTS mobile-phone telecommunication systems (Bertacco et al., 2004).

The LOP-CC is $\mathcal{NP}$-hard (Bertacco et al., 2004).

In the LOP-CC

- a weight $p_i \geq 0$ is given $\forall i \in \mathcal{N}$;
- a cost $c_{ij} \geq 0$ is given $\forall (i,j) \in \mathcal{A}$.

Remark: In general $c_{ij} \neq c_{ji}$.

The objective function to be minimized is the sum of cost terms α associated with the vertices and recursively defined in this way:

$$\alpha_i = p_i + \sum_{j|\Pi^{-1}(j)>\Pi^{-1}(i)} c_{ij}\alpha_j \quad \forall i \in \mathcal{N}.$$

A Non-linear Integer Programming model of the LOP-CC

$$\min \; z = \sum_{i \in \mathcal{N}} \alpha_i$$

$$\begin{aligned}
\text{s.t.} \quad & \alpha_i = p_i + \sum_{j=1}^{N} c_{ij} x_{ij} \alpha_j && \forall i \in \mathcal{N} && (1)\\
& x_{ij} + x_{ji} = 1 && \forall i, j \in \mathcal{N} && (2)\\
& x_{ij} + x_{jk} + x_{ki} \leq 2 && \forall i, j, k \in \mathcal{N} && (3)\\
& \alpha_i \in \Re_+ && \forall i \in \mathcal{N} &&\\
& x_{ij} \in \{0, 1\} && \forall i, j \in \mathcal{N}. &&
\end{aligned}$$

The objective

The objective is the minimization of the overall value of variables α that represent the costs accumulated at the vertices.

$$\text{minimize} \quad z = \sum_{i \in \mathcal{N}} \alpha_i$$

The value of each term α_i is given by constraints

$$\alpha_i = p_i + \sum_{j=1}^{N} c_{ij} x_{ij} \alpha_j \quad \forall i \in \mathcal{N}$$

and depends on the cumulative costs α of the vertices that follow i in the acyclic tournament defined by the x variables.

The constraints are non-linear.

The constraints

Constraints

$$x_{ij} + x_{ji} = 1 \quad \forall i, j \in \mathcal{N}$$

$$x_{ij} + x_{jk} + x_{ki} \leq 2 \quad \forall i, j, k \in \mathcal{N}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in \mathcal{N}$$

impose that the values of the binary variables x define an acyclic tournament.

Branch-and-bound

Consider a branch-and-bound algorithm, in which each position is assigned a vertex, from position N down to position 1.
Branching is on an integer variable: at level l, $N - l$ sub-problems are generated.

For each path in the decision tree, all values of α variables are recursively computed, from $\alpha_{\Pi(N)}$ down to $\alpha_{\Pi(1)}$.

In order not to enumerate all the 2^N possible permutations, the algorithm exploits a lower bounding technique.

Lower bounding

Consider the generic node at level l in the search tree (the root is at level 0):
l vertices have been assigned to the last l positions in the acyclic tournament;
the other vertices are still to be ordered.

Call $\mathcal{L}$ the set of the unordered vertices and $\mathcal{R}$ the set of the vertices already ordered.

Example

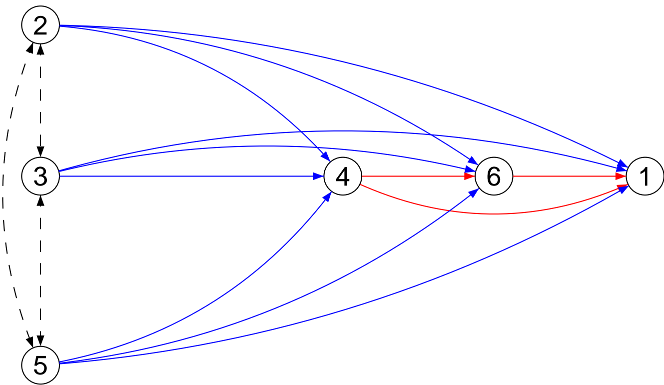

Figure: A small example with six vertices, three fixed and three to be decided.

Lower bounding

The value of the objective function can be split into two a left and a right term:

$$z = \sum_{i \in \mathcal{L}} \alpha_i + \sum_{i \in \mathcal{R}} \alpha_i$$

The right term can be computed exactly, since the vertices in $\mathcal{R}$ have already been ordered.
The left term can be split into three terms:

$$\sum_{i \in \mathcal{L}} \alpha_i = \sum_{i \in \mathcal{L}} \sum_{j \in \mathcal{L}} c_{ij} x_{ij} \alpha_j + \sum_{i \in \mathcal{L}} p_i + \sum_{i \in \mathcal{L}} \sum_{j \in \mathcal{R}} c_{ij} \alpha_j$$

For each $i \in \mathcal{L}$ the left term includes

- the contributions to α_i due to the vertices in $\mathcal{L}$,
- the fixed cost
- the contributions to α_i due to the vertices in $\mathcal{R}$.

A valid lower bound

The term $\sum_{i \in \mathcal{L}} \sum_{j \in \mathcal{R}} c_{ij}\alpha_j$ does not depend on the x variables since all vertices in $\mathcal{R}$ follow all vertices in $\mathcal{L}$ in the acyclic tournament under construction (i.e. $x_{ij} = 1 \; \forall i \in \mathcal{L}, j \in \mathcal{R}$).

Hence the values of $\sum_{i \in \mathcal{L}} p_i$ and $\sum_{i \in \mathcal{L}} \sum_{j \in \mathcal{R}} c_{ij}\alpha_j$ can be computed exactly.

A valid lower bound is given by

$$LB_0 = \sum_{i \in \mathcal{L}} p_i + \sum_{i \in \mathcal{L}} \sum_{j \in \mathcal{R}} c_{ij}\alpha_j + \sum_{i \in \mathcal{R}} \alpha_i.$$

We define

$$H = z - LB_0 = \sum_{i \in \mathcal{L}} \sum_{j \in \mathcal{L}} c_{ij} x_{ij} \alpha_j$$

$H \geq 0$, because $p \geq 0$, $c \geq 0$ and hence $\alpha \geq 0$.
Any lower bound to H can be used to strengthen LB_0.

Improving the lower bound

We define $\beta_i = p_i + \sum_{j \in \mathcal{R}} c_{ij}\alpha_j \quad \forall i \in \mathcal{L}$.

In any solution α_i is certainly not less than β_i at any intermediate node in the search tree, owing to the non-negativity of the data.

Therefore β_i is a valid lower bound to the final value of α_i.

Improving the lower bound

For each pair of distinct vertices i and j in $\mathcal{L}$, we know that in a complete solution either i precedes j or vice versa: therefore either $c_{ij}\alpha_j$ or $c_{ji}\alpha_i$ contributes to the overall cost.

Hence the following inequalities hold:

$$H \geq \sum_{i,j \in \mathcal{L}: i<j} \min\{c_{ij}\alpha_j, c_{ji}\alpha_i\} \geq \sum_{i,j \in \mathcal{L}: i<j} \min\{c_{ij}\beta_j, c_{ji}\beta_i\}.$$

Hence the value

$$LB_1 = LB_0 + \sum_{i,j \in \mathcal{L}: i<j} \min\{c_{ij}\beta_j, c_{ji}\beta_i\}$$

is a valid lower bound.

LB_1 is computable in $O((N-l)^2)$ at each node at level l in the search tree.

Computational results

The computational tests refer to four data-sets, each made of 500 randomly generated instances with $N = 16, 14, 12$.

For each set of 500 instances and for each size we observed:

- the average and the maximum computing time
- the average and the maximum number of nodes explored in the search tree.

These values are reported for the branch-and-bound algorithm with lower bound LB_0 and LB_1.

The search tree was explored depth-first.

Computational results

Set	Size	Average time LB_0	Average time LB_1	Max. time LB_0	Max. time LB_1	Avg. n. nodes LB_0	Avg. n. nodes LB_1	Max. n. nodes LB_0	Max. n. nodes LB_1
A	12	0.03	0.00	1.00	0.00	138249.65	1815.98	3419066	46409
	14	0.50	0.01	31.00	1.00	2745671.10	11862.01	166581704	529724
	16	12.58	0.14	973.00	5.00	60974332.11	100422.75	4729842076	4963008
B	12	0.05	0.01	2.00	1.00	270668.24	6806.23	12487041	260226
	14	1.30	0.07	105.00	3.00	7037065.99	19438.65	562082006	4077536
	16	38.85	1.50	2022.00	178.00	188009341.31	137804.72	9804359196	204379303
C	12	0.02	0.00	1.00	0.00	151073.13	5005.30	3820123	111448
	14	0.53	0.06	29.00	3.00	2890538.03	52734.75	160998680	3273061
	16	14.94	1.00	1069.00	44.00	72449144.17	740194.33	5178947760	37629119
D	12	0.03	0.00	1.00	1.00	170845.18	5907.80	3338684	140596
	14	0.64	0.07	16.00	2.00	3488891.31	67846.74	89150125	1930878
	16	16.78	1.20	542.00	33.00	81387534.69	861567.37	2623680651	26285589
Avg.	12	0.03	0.00	1.25	0.05	148734.07	4883.83	5766228.50	139669.75
	14	0.74	0.05	45.50	2.25	4040541.61	37970.54	244703128.75	2452799.75
	16	20.79	0.96	1151.50	65.00	100705088.07	459997.29	5584207420.75	68314254.75

Table: Comparison between B&B algorithms using LB_0 and LB_1.

Comparison at the root node

Data-set	Size	LB_0	LB_1
A	12	44.64 %	15.23 %
	14	51.39 %	20.42 %
	16	57.22 %	25.75 %
B	12	69.73 %	37.76 %
	14	76.39 %	47.09 %
	16	82.30 %	56.49 %
C	12	69.67 %	40.54 %
	14	77.03 %	50.62 %
	16	83.09 %	60.24 %
D	12	73.90 %	45.23 %
	14	81.02 %	56.27 %
	16	86.52 %	65.92 %
Average	12	64.49 %	34.69 %
	14	71.46 %	43.60 %
	16	77.28 %	52.10 %

Table: Average percentage gaps at the root node ($\frac{z^* - LB}{LB}$).

Search strategy

Depth-first-search allows for an effective recursive implementation of the code: the vertices are initially sorted according to their fixed cost p_i and this sequence determines the order in which sub-problems are visited.

The unordered vertices (those in $\mathcal{L}$) are kept in a FIFO queue and each step down in the search tree corresponds to a *Pop* operation, while each backtrack step corresponds to a *Push* operation on the queue.

In this way the first leaf that the algorithm examines corresponds to the solution in which the vertex with the smallest fixed cost is in the last position (i.e., it is fixed first) and the one with the largest fixed cost is in the first position (i.e., it is fixed last).

Search strategy

However the leaves of the search tree are not visited according to the lexicographic order defined by the initial sequence.

For instance, if the starting sequence is $\langle 1, 2, 3, 4 \rangle$, with $p_1 > p_2 > p_3 > p_4$, the leaves of the search tree are generated in this order:

- $\langle 1, 2, 3, 4 \rangle$
- $\langle 2, 1, 3, 4 \rangle$
- $\langle 3, 1, 2, 4 \rangle$
- $\langle 1, 3, 2, 4 \rangle$
- ...and so on.

Then, $\langle 3, 1, 2, 4 \rangle$ precedes $\langle 1, 3, 2, 4 \rangle$ even if $p_1 > p_3$.

Search strategy

Even if this technique is very effective from a software implementation viewpoint, a suitable sorting of the successor nodes, repeated at each node of the search tree, may allow the algorithm to discover the optimal solution earlier.

To have a quantitative figure of merit of the effectiveness of the branching policy and in particular of the ordering in which children nodes are explored, it is necessary to measure “how early” the optimal solution is discovered.

For this purpose, let us consider a generic instance of the LOP-CC and the corresponding search tree as it is explored by a depth-first-search branch-and-bound algorithm and let us indicate by P^* the optimal path to be followed along the search tree to reach the first leaf corresponding to an optimal solution.

Evaluating the search strategy

Let us define a vector b_l with one component for each level l of the search tree. Each component b_l may range in $[0, \ldots, N - l - 1]$, that is it may take on a number of different values equal to the number of successor nodes at that level.

The value of b_l indicates the number of “wrong” branches that have been explored from the node of P^* at level l before the branch that leads to the optimal solution.

A vector $b = (0, 0, \ldots, 0)$ represents the ideal case, in which the algorithm finds the optimal solution as the first leaf of the search tree.

Dividing b_l by the maximum number of successor nodes at level l, we get a fraction $\eta_l = \frac{b_l}{N-l-1}$, whose value is between 0 and 1, representing the effectiveness of the branch sorting policy at level l.

A new search strategy

We measured the average values of the components of vector η over all the instances of the benchmark, and we observed rather high values, often greater than 0.5.

This motivated the design of a different policy to sort the successors, inspired by lower bound $LB1$.

For each pair of vertices $i, j \in \mathcal{L}$, a choice is made between arcs (i, j) and (j, i), choosing the minimum among the two costs $c_{ij}\beta_j$ and $c_{ji}\beta_i$: if $c_{ij}\beta_j < c_{ji}\beta_i$ we tentatively orient the arc from i to j.

Depending on these decisions the in-degree of each vertex is determined.

Then all the vertices in $\mathcal{L}$ are sorted by increasing values of their in-degree so that the one with maximum in-degree is the last one in the ordering and then it is fixed first.

Computational results

Level	FIFO queue η	Sorted branches η
1	0.034	0.034
2	0.267	0.045
3	0.375	0.049
4	0.370	0.046
5	0.408	0.040
6	0.373	0.037
7	0.379	0.025
8	0.330	0.027
9	0.313	0.029
10	0.321	0.031
11	0.317	0.019
12	0.254	0.020
13	0.242	0.017
14	0.179	0.010
15	0.180	0.000
16	0.000	0.000
Avg. n. nodes	100422.75	96265.056
Max. n. nodes	4963008	5177634
Avg. time	0.134	0.122
Max. time	4	5

Table: Effects of branch sorting (avg. on 500 instances).

The overall computing time is the same, but η is smaller, i.e. the algorithm finds optimal solutions earlier.
This suggests a criterion to truncate the exploration of the search tree.

Truncated branch-and-bound

Time is very critical for the UMTS application from which the problem arises. Therefore it is important to develop very fast heuristics still providing good solutions.

Faster results can be obtained by truncating the branch-and-bound algorithm in order to reduce the computing time at the expense of the optimality guarantee.

Truncated branch-and-bound

For this purpose the fraction of branches to be explored at each level of the search tree is limited by a given threshold $0 \leq \theta \leq 1$: at each node of the search tree the successors nodes are sorted as explained above and the algorithm explores only the first $\lfloor \theta(N - I - 1) \rfloor + 1$ of them.

Setting $\theta = 1$ the whole search tree is (implicitly) explored and the best solution found is guaranteed to be optimal.

Setting $\theta = 0$ only the first branch is explored at each node and the search tree reduces to a path from the root to the first leaf.

Choosing intermediate values one can tune the trade-off between speed and solution cost.

Computational results

Size	θ	Avg. time	Max. time	Avg. gap	Max. gap	Opt.sol.
12	0.00	0.000	0.000	8.764 %	147.793 %	232
	0.25	0.000	0.000	1.018 %	37.446 %	1303
	0.50	0.002	1.000	0.053 %	7.627 %	1913
	0.75	0.002	1.000	0.007 %	3.306 %	1991
	1.00	0.005	1.000	0.000 %	0.000 %	2000
14	0.00	0.001	1.000	12.539 %	140.787 %	95
	0.25	0.003	1.000	1.303 %	37.770 %	1155
	0.50	0.012	1.000	0.097 %	10.062 %	1903
	0.75	0.025	1.000	0.011 %	2.978 %	1990
	1.00	0.053	3.000	0.000 %	0.000 %	2000
16	0.00	0.000	0.000	20.355 %	369.291 %	34
	0.25	0.014	1.000	1.697 %	37.544 %	947
	0.50	0.129	14.000	0.378 %	10.387 %	1846
	0.75	0.358	56.000	0.005 %	1.897 %	1990
	1.00	0.903	173.000	0.000 %	0.000 %	2000

Table: Truncated branch-and-bound.

For $\theta = 0.25$ optimal solutions are found for about 50% of the instances.
For $\theta = 0.50$ optimal solutions are found for about 95% of the instances.
The speed-up can be up to a factor of 3 for $\theta = 0.75$ and a factor of 10 for $\theta = 0.50$.

Large instances

The algorithms were also tested on larger instances with the purpose of studying which is the maximum size for which an approach based on truncated branch-and-bound is viable.

The data-set consists of sub-matrices of different size taken from a randomly generated (32×32) cost matrix.

It was reported that instances with up to $N = 20$ could be solved in three hours using LB_0.
Using LB_1 we could solve instances with up to $N = 26$ within one hour and a half.
A trial with $N = 28$ was stopped after 24 hours.

Exact and truncated B&B: large instances

Size	Exact		Truncated		
	Time	Cost	Time	Cost	θ
18	0:00:02	5.636060	0:00:01	5.636060	0.50
20	0:01:31	8.115764	0:00:38	8.115764	0.50
22	0:03:10	8.178600	0:01:17	8.178600	0.50
24	0:16:59	11.962475	0:06:17	11.962475	0.50
26	1:37:33	15.208841	0:02:31	15.208841	0.25
28	$>$24:00:00		0:53:54	17.428847	0.25

Table: Results obtained on larger instances.

Constructive heuristics

The truncated branch-and-bound can be easily turned into a fast constructive heuristic by setting $\theta = 0$.

We compare it with a GRASP heuristic (Benvenuto et al. 2005) and a tabu search (Duarte et al. 2006).

All data-sets, made of 25 instances each, were solved in less than one second on a 2.0 GHz processor.
Computing times for GRASP and TSLOPCC were measured on a 3.2 GHz processor.

Comparison with other heuristics

Size	GRASP		Constr. heur.	TSLOPCC	
	Cost	Time	Cost	Cost	Time
35	0.935	1.68	0.588	0.344	0.39
100	8.48 E+05	24.75	1.43 E+04	1.20 E+03	30.75
150	2.90 E+11	78.31	8.84 E+07	2.25 E+06	180.43

Table: Comparison of heuristics on random large instances.

The constructive heuristic obtained from the truncated branch-and-bound algorithm dominates GRASP in both time and cost.

No domination exists between truncated branch-and-bound and tabu search: the former is faster, the latter is more accurate.