Low-Memory Tour Reversal in Directed Graphs

Viktor Mosenkis[1], Uwe Naumann[1], Elmar Peise[1]

LuFG Informatik 12, RWTH Aachen University
52056, Aachen, Seffenter Weg 23, Germany
mosenkis@stce.rwth-aachen.de

We consider the problem of reversing a *tour* $(i_1, i_2, \ldots, i_l)$ in a directed graph $G = (V, E)$ with positive integer vertices V and edges $E \subseteq V \times V$, where $i_j \in V$ and $(i_j, i_{j+1}) \in E$ for all $j = 1, \ldots, l-1$. The tour can be processed in last-in-first-out order as long as the size of the corresponding stack does not exceed the available memory. This constraint is violated in most cases when considering control-flow graphs of large-scale numerical simulation programs. The tour reversal problem also arises in adjoint programs used, for example, in the context of derivative-based nonlinear optimization, sensitivity analysis, or other, often inverse, problems. The intention is to compress the tour in order not to run out of memory. As the general optimal compression problem was proven to be NP-hard [1] and big control-flow graphs results from loops in programs we do not want to use general purpose algorithms to compress the tour. We rather want to compress the tour by finding loops and replace the redundant information by proper representation of the loops.

Definition 1 *A* compressed tour $C = (N, L)$ *in a directed graph* $G = (V, E)$ *consists of a stack of integers* $N = (i_1, i_2, \ldots, i_s)$ *where* $i_j \in V$ *for* $j = 1, \ldots, s$ *and a stack of loops* $L = (l_1, l_2, \ldots, l_p)$. *A loop* l_k *contains an entry index* $i^- \in \{1, \ldots, s\}$, *an exit index* $i^+ \in \{1, \ldots, s\}$, *such that* $i^- \leq i^+$, *and an integer* $m_k > 0$ *holding the loop's multiplicity. For any two loops* (i_1^-, i_1^+, m_1) *and* (i_2^-, i_2^+, m_2), *w.l.o.g.* $i_1^- \geq i_2^-$, *we require disjointness or inclusion, that is*

$$i_1^+ < i_2^- \quad \vee \quad \left(i_2^- \leq i_1^- \wedge i_1^+ \leq i_2^+\right) \wedge (i_1^-, i_1^+) \neq (i_2^-, i_2^+).$$

$(i_1^-, i_1^+) \neq (i_2^-, i_2^+)$ *is needed to ensure, that we do not have two equal loops in* L.

$$f(C) := |N|$$

is the cost function of this compressed tour.

A compressed tour $C = (N, L)$ in a directed graph G is decompressed by unrolling all loops in L in LIFO order recursively. C is called a compressed version of a tour T, if and only if the tour that results from the decompression of C is equal to T.

As the problem of finding a compressed version of a tour T with minimal cost exhibits the properties of overlapping subproblems and optimal substructure, an optimal representation of a tour, exploiting the compression of loops, can be found with the help of dynamic programming.

Dagstuhl Seminar Proceedings 09061
Combinatorial Scientific Computing
http://drops.dagstuhl.de/opus/volltexte/2009/2092

The required concatenation operation $\circ$ for two stacks s_1 and s_2 is defined as follows: Let

$$s_1 = \left[s'_1, \underbrace{\begin{matrix} i^1_1, \ldots, i^1_n \\ l^1_1, \ldots, l^1_u \end{matrix}}_{s''_1} \right], \quad \text{and } s_2 = \left[\underbrace{\begin{matrix} i^2_1, \ldots, i^2_n \\ l^2_1, \ldots, l^2_u \end{matrix}}_{s''_2}, s'_2, \right]$$

where s'_1 and s'_2 denote arbitrary substacks of s_1 and s_2, respectively, and l^j_1 always indicates the loop from i^j_1 to i^j_l, $j \in \{1, 2\}$. If such a loop does not exist, then m^j_1 is set to one. In the case where s''_1 and s''_2 vary only in m^1_1 and m^2_1

$$s = s_1 \circ s_2 \equiv \left[s'_1, \begin{matrix} i^1_1, \ldots, i^1_n \\ l, l^1_2, \ldots, l^1_u, \end{matrix}, s'_2 \right], \quad \text{where } l = (i^{-1}{}_1, i^{+1}{}_1, m^1_1 + m^2_1) \quad .$$

An optimally compressed version of the subtour of T ranging from the i-th to the j-th entry is denoted by $d_{i,j}$. The offline dynamic programming algorithm computes $d_{i,j}$ according to

$$f(d_{i,j}) = \begin{cases} 1 & i = j \\ \min\limits_{i \le s < j} f(d_{i,s} \circ d_{s+1,j}) & \text{otherwise} \end{cases} \quad .$$

The offline algorithm has two disadvantages. One needs to store the whole tour first before the compression is started. But the uncompressed tour may exceed the memory. The other problem is runtime. Dynamic programming yields a cubic computational complexity. It is likely to be inefficient for real problems. Hence, we look for good and fast online heuristics.

Compression should take place as soon as possible. The algorithm works on a window of a predefined size s. To find loops we take a naive approach. Once a new element has been added to the stack we start to search for loops of length $1, \ldots, l$, where $l < s/2$, in increasing order beginning from the new element. The stack is compressed as soon as a loop is found by adding a corresponding loop entry to the compressed tour followed by removing integers that are no longer needed from the stack. The window is then refilled with entries on top of the stack. The refilling of the window is important in order to be able to find loops whose compressed version contains less than $s/2$ entries while their uncompressed version is longer than the size of the window.

One can show that the online algorithm produce near-optimal results in mostly relevant cases.

To decompress the compressed tour we unroll loops on top of the stack, for as long as the the element on top of the stack is not a value element. This method saves memory, and allows us to get away with the same amount of memory as during the compression.

The Table 1 shows the compression rates that are achieved by applying the online algorithm to the flow of control of three representative test problems.

Table 1. Compressions rates produced by online algorithm with window size 21 applied to control-flow stacks of the solid fuel ignition (sfi) [2], burger and Leonard Jones cluster (ljc) [2] problems

Problem	uncompr. stack size	compr. stack size	compr. rate
burger	3432 MB	144 Byte	25013889
sfi	4576 MB	168 Byte	28572380
ljc	3692 MB	703984 Byte	5500

References

1. Storer, A., Szymanski, T.G.: Data compression via textual substitution. Journal of the Association for Computing Machinery **29** (1982) 928–951
2. Averik, B., Cartere, R., Moré, J.: The Minpack-2 test problem collection (preliminary version). Technical Report 150, Mathematical and Computer Science Division, Argonne National Laboratory (1991)