Pergamon

Computers Math. Applic. Vol. 27, No. 9/10, pp. 53–58, 1994

0898-1221/94 $7.00 + 0.00

0898-1221(94)E0048-O

Dynamic Programming and Graph Optimization Problems

T. C. Hu and J. D. Morgenthaler
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114, U.S.A.

Abstract—Several classes of graph optimization problems, which can be solved using dynamic programming, are known to have more efficient tailor-made algorithms. This paper discusses four such classes and the underlying constraints on their subproblem interrelationships that yield these efficient algorithms. These classes are also extended to handle more general cost functions.

1. INTRODUCTION

Dynamic programming is a particular way of thinking in problem-solving, just as mathematical induction is a particular way of proving theorems. Over 40 books (for example [1–6]) and thousands of papers have been written on the subject, making the impact of Bellman's contribution on many diverse fields impossible to trace (e.g., [7]).

In the present paper, we shall discuss four classes of graph optimization problems which can all be solved using dynamic programming. These classes are

(1) Optimum Path Problems
(2) Optimum Binary Tree Problems
(3) Triangulation of a Polygon
(4) Network Partitioning.

Because dynamic programming is such a general principle and can be applied to many problems, tailor-made algorithms can be more efficient than algorithms based on dynamic programming alone. We shall discuss such tailor-made algorithms and let the reader decide what special structures make these improvements possible.

2. OPTIMUM PATH PROBLEMS

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ with vertices or nodes $v_i \in \mathcal{V}$, where $i = (1, 2, \ldots, n)$ and directed arcs a_{ij} connecting v_i to v_j. Length d_{ij} is associated with every arc $a_{ij} \in \mathcal{A}$. We do not require that $d_{ij} \geq 0$ or that $d_{ij} = d_{ji}$, but we assume that there exists no negative length cycle. The problem is to find the shortest paths between all pairs of nodes in the graph.

In terms of the shortest path problem, the principle of optimality would be

> If v_i and v_j are two intermediate nodes in a shortest path, then the subpath from v_i to v_j must be a shortest path from v_i to v_j.

Using this principle recursively, we see that there must be some pairs of nodes, say v_k and v_l, for which the shortest path consists of the single arc a_{kl}. Such arcs are called *basic arcs*.

Typeset by $\mathcal{A}_\mathcal{M}\mathcal{S}$-TeX

The classical Floyd-Warshall shortest path algorithm [8,9] sums up the total length of a shortest path from v_a to v_z consisting of basic arcs and creates a new basic arc a_{az} with length equal to the sum of the lengths of those basic arcs. The Floyd-Warshall algorithm can be stated as

$$d_{ik} := \min(d_{ik}, d_{ij} + d_{jk}), \qquad \text{for } j = 1, \dots, n, \quad \text{and } i, k \neq j, \tag{1}$$

and it has a very clever way of keeping track of the intermediate nodes. Operation (1) is called the triple operation in [10], since it involves three arcs. The operations min and + can be replaced by others which maintain a *closed semiring*, as shown in [11].

For example, in other optimum path problems, we may want to define the value of a path to be

$$L(a_{ab}, a_{bc}, \dots, a_{yz}) = \max(d_{ab}, d_{bc}, \dots, d_{yz}), \tag{2}$$

and seek the path of minimum value. We can solve these problems by modifying the operation in (1) to be

$$d_{ik} := \min\left(d_{ik}, \max(d_{ij}, d_{jk})\right), \qquad \text{for } j = 1, \dots, n, \quad \text{and } i, k \neq j. \tag{3}$$

A natural question is, "What other optimum path problems could be solved by further modification of the triple operation?"

Using $a_1, a_2, \dots, a_m$ to represent the values of m arcs of a path, we can generalize the Floyd-Warshall algorithm to handle optimum path problems whose values are defined appropriately [12]. For four arcs, with L as the generalized length function, we only require that

$$\begin{aligned} L(a_1, a_2, a_3, a_4) &= L[L(a_1, a_2), L(a_3, a_4)], \\ &= L[L(a_1), L(a_2, a_3, a_4)], \\ &= L[L(a_1, a_2, a_3), L(a_4)], \end{aligned} \tag{4}$$

and

$$L(a_i, a_j) \leq L(a_k, a_l), \qquad \text{if } a_i \leq a_k,\ a_j \leq a_l. \tag{5}$$

3. OPTIMUM BINARY TREES

In this class of problems, we are given a set of square nodes $\mathcal{V}$ with a weight $w_i > 0$ associated with each square node $v_i \in \mathcal{V}$. We wish to construct a binary tree with these square nodes as the leaves. To differentiate them from the square nodes, the internal nodes of this tree are called *circular* nodes. The problem is to construct a binary tree such that the sum of the weighted paths from the root to all the leaves is minimum. Letting l_i be the number of edges in the path from the root to leaf v_i, we can state our goal as $\min \sum_i w_i\, l_i$.

Here, the dynamic programming principle would be

> Any subtree of an optimum tree must be an optimum tree for the set of leaves of that subtree.

In general, not every pair of square nodes can be combined to form a subtree of two leaves. We start with the graph $\mathcal{G}^* = (\mathcal{V}, \mathcal{E})$, called the *underlying constraint graph*, which shows all allowable node combinations. If square nodes v_i and v_j are allowed to combine to form a subtree rooted at the new circular node $v_{i,j}$, then the underlying constraint graph contains edge e_{ij}. Adjacency in this graph is inherited by the circular nodes in the binary tree as the tree is constructed from the leaves (square nodes) in bottom-up fashion. When (square or circular) nodes v_i and v_j are combined, parent $v_{i,j}$ inherits all adjacent nodes from both v_i and v_j, creating a new condensed constraint graph that shows further allowable combinations.

If the underlying constraint graph is a complete graph, then the nodes are free to be combined in any way. In this case, we can use the classical Huffman's algorithm [13] which always combines the two nodes with the smallest weights w_i and w_j and assigns their parent (new circular node $v_{i,j}$) the weight $w_i + w_j$. If the underlying constraint graph is a chain, then we have the optimum alphabetic tree problem. Here we can use the Hu-Tucker algorithm [14,15] with time complexity $O(n \log n)$.

If the underlying constraint graph is an arbitrary graph, we could in principle, successively construct optimum subtrees and merge them into an optimum tree with all leaves corresponding to the square nodes. However, the work to construct the table could be prohibitive.

A natural question is, "What cost functions enable us to use the same procedures as Huffman and Hu-Tucker?" This was answered in [16] for a class of functions called regular functions.

In other applications [17], the cost function may be very similar to the cost function of constructing a binary tree, namely

$$c_{ik} = \min_j \left[c_{ij} + c_{(j+1)k}\right] + w_{ik}, \tag{6}$$

where c_{ik} is the cost of the subtree containing square nodes v_i through v_k, and $w_{ik} = \sum_{j=i}^{k} w_j$. If the w_{ik} satisfy certain conditions, say

$$w_{ab} + w_{cd} \le w_{bc} + w_{ad}, \tag{7}$$

which in turn implies

$$c_{ab} + c_{cd} \le c_{bc} + c_{ad}, \tag{8}$$

then the straight $O(n^3)$ dynamic program can be made $O(n^2)$ as shown by Knuth [18] and Yao [17].

This kind of min-cost alphabetic binary tree is, in a sense, a generalization of a binary search tree used to search for an item from among n items already sorted alphabetically on a tape. If we add the cost of moving a "read head" along the tape, in addition to the cost of comparisons, then we need a hybrid of a complete binary search tree and a linear sequential search tree [19].

4. POLYGON TRIANGULATION

Given a convex polygon $\mathcal{P}$ with n sides, a "partitioning" of $\mathcal{P}$ into $n-2$ nonoverlapping triangles whose vertices are vertices of $\mathcal{P}$ is called a triangulation or tiling, and every triangle is a tile. Every possible tile has a given arbitrary cost. The problem is to find a tiling of $\mathcal{P}$ such that the sum of the costs of the tiles used is minimum. The name "tiling" comes from the intuitive meaning of tiling the floor of a convex room with triangular tiles.

For polygon triangulation, we can state the principle of optimality as:

> Any subpolygon of an optimally partitioned convex polygon must be partitioned optimally.

The problem can be formulated as a linear programming problem with special structure such that the extreme feasible solutions all have integer components. The special structure of the matrix of this linear program enables us to develop a recursive $O(n^3)$ algorithm [20].

An interesting special case of this problem is to find the order of multiplication of n rectangular matrices that minimizes the total number of multiplications. Here, a matrix with dimensions $p \times q$ is represented by an edge whose two end vertices have weights p and q. The matrix chain is represented by a sequence of edges $(p \times q), (q \times r), (r \times s), (s \times t) \cdots (y \times z)$ and the resulting matrix is of dimension $p \times z$. Thus, the matrix chain of $n-1$ matrices is represented by a polygon of n edges, where the multiplication of a $(p \times q)$ matrix with a $(q \times r)$ matrix is associated with a triangle cost of $(p \cdot q \cdot r)$. We have an n-sided convex polygon, where every vertex has weight w_i, and triangle $v_i v_j v_k$ has cost $(w_i \cdot w_j \cdot w_k)$. A straight dynamic programming table build-up would result in a $O(n^3)$ algorithm, but a special tailor-made algorithm yields $O(n \log n)$ [21,22], and a linear algorithm with error bound [23,24].

In a sense, the polygon triangulation problem is somewhat like the alphabetic binary tree problem, except that the "underlying constraint graph" is a cycle instead of a chain. Also, the elementary object is an edge with weights w_i and w_j, not a node. When every node has two weights (a left weight and a right weight), we can also construct an optimum binary tree in $O(n^2)$ time [25].

5. NETWORK PARTITIONING

Given a graph $\mathcal{N} = (\mathcal{V}, \mathcal{A})$, called a *network*, with n nodes and where every arc $a_{ij} \in \mathcal{A}$ has a positive integer capacity $c_{ij} \equiv c_{ji}$ [26]. The multiterminal flow problem [27] is to determine the maximum flow between every pair of nodes in the network. Due to the Max-Flow Min-Cut theorem [28,29], the maximum flow value between each of the $\binom{n}{2}$ different pairs of nodes is equal to the minimum cut separating that pair of nodes. A cut is a partitioning of $\mathcal{V}$ into two proper subsets X and $\overline{X}$, where the value (capacity) of the cut is defined to be

$$C(X, \overline{X}) = \sum c_{ij}, \qquad \text{where } (i \in X,\ j \in \overline{X}).$$

As shown in [27], a subset of $n-1$ *noncrossing* cuts among the $\binom{n}{2}$ minimum cuts is enough to determine all $\binom{n}{2}$ maximum flows. Two cuts $(X, \overline{X})$ and $(Y, \overline{Y})$ cross each other if each of the four sets $X \cap Y$, $X \cap \overline{Y}$, $\overline{X} \cap Y$, and $\overline{X} \cap \overline{Y}$ is nonempty. A set of cuts is *non-crossing* if no two cuts in the set cross each other.

It can be shown that there is a one-to-one correspondence between any $n-1$ noncrossing cuts and a tree. The so-called Gomory-Hu cut tree corresponds to such a set of $n-1$ noncrossing cuts where the total sum of the cut values is a minimum.

In this case, the dynamic programming principle would be

> If a cut $(X, \overline{X})$ is selected to be one of the tree cuts partitioning $\mathcal{V}$, then the subsets X and $\overline{X}$ must be partitioned optimally.

Only $n-1$ maximum flow computations [30,31] are needed to determine the $n-1$ minimum cuts of the Gomory-Hu cut tree. Once we have this tree, the minimum cut partitioning v_i and v_j is simply the tree arc with the minimum value on the unique path from v_i to v_j in the cut tree.

In many applications, the value of a cut $(X, \overline{X})$ is defined differently [32–34]. For example, we can define the value to be

$$C(X, \overline{X}) = \sum_{(i \in X,\, j \in \overline{X})} \frac{c_{ij}}{|X| \cdot |\overline{X}|}.$$

We may still want to find the $n-1$ minimum cuts which separate the $\binom{n}{2}$ different pairs of nodes.

However, there are $2^{n-1} - 1$ cuts in an n-node network. A straight-forward dynamic programming algorithm would require too much time and space.

Assume the values of all cuts are defined *arbitrarily*, and we have a subroutine which can find the minimum cut separating a given pair of nodes. How many times do we have to call this subroutine so that we know the minimum cut separating any pair of nodes? The answer is that we need to use the subroutine only $n-1$ times to construct a tree where each internal node of the tree corresponds to a cut, and every leaf of the tree corresponds to a node in the original network. The minimum cut separating a leaf v_i and a leaf v_j is the least common ancestor of v_i and v_j [35].

6. FINAL REMARKS

Dynamic programming is based on a simple and yet profound idea which cannot be totally formalized. This makes it unlike greedy algorithms, which are based on the theory of greedoids [36]. In a nutshell, dynamic programming is the art of decomposing a complex problem into subproblems and combining the optimum solutions to these subproblems without duplication of computations. The success of dynamic programming lies in the fact that an optimum solution to a subproblem usually depends only on the optimum values of adjacent subproblems and not on the structure of these adjacent subproblems.

In graph terminology, we may consider an elementary subproblem as a node v_i, and the computational effort to solve the subproblem as weight w_i. The interrelationship between the subproblems is the underlying constraint graph showing which subproblems (nodes) can be combined. The final goal is to successively cluster all the nodes of the underlying constraint graph into one node.

Recent efforts to implement dynamic programming algorithms in the framework of parallel computation [37–39] will undoubtedly open another horizon for dynamic programming. In closing, the following caption from the chapter on dynamic programming in [10] is dedicated to the memory of Dr. Richard E. Bellman.

Live optimally today, for today is the first day of the rest of your life.

REFERENCES

1. R. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, NJ, (1957).
2. R. Bellman and S.E. Dreyfuss, *Applied Dynamic Programming*, Princeton University Press, Princeton, NJ, (1962).
3. E.V. Denardo, *Dynamic Programming: Models and Applications*, Prentice-Hall, Englewood Hills, NJ, (1982).
4. S.E. Dreyfuss and A.M. Law, *The Art and Theory of Dynamic Programming*, Academic Press, New York, (1977).
5. G.L. Nemhauser, *An Introduction to Dynamic Programming*, John Wiley & Sons, New York, (1966).
6. M. Sniedovich, *Dynamic Programming*, Pure and Applied Mathematics: A Series of Monographs and Textbooks, Marcel Dekker, New York, (1992).
7. A. Lew, Richard Bellman's contribution to computer science, *Math Analysis and Applications* **119** (1/2), 90–96 (1986).
8. R.W. Floyd, Algorithm 97, Shortest path, *Communications of the ACM* **5**, 345 (1962).
9. S. Warshall, A theorem on Boolean matrices, *Journal of the ACM* **9**, 11–12 (1962).
10. T.C. Hu, *Combinatorial Algorithms*, Addison-Wesley, Reading, MA, (1982).
11. A.V. Aho, J.E. Hopcraft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, (1974).
12. V. Klee and D. Larman, Use of Floyd's algorithm to find shortest restricted paths, *Annals of Discrete Mathematics* **4**, 237–249 (1979).
13. D.A. Huffman, A method for the construction of minimum redundancy codes, *Proceedings of the IRE* **40**, 1098–1101 (1952).
14. T.C. Hu and A.C. Tucker, Optimal computer search trees and variable-length alphabetic codes, *SIAM Journal on Applied Mathematics* **21** (4), 514–532 (1971).
15. D.E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching*, Addison-Wesley, Reading, MA, (1973).
16. T.C. Hu, D.J. Kleitman and J.K. Tamaki, Binary trees optimum under various criteria, *SIAM Journal on Applied Mathematics* **37** (2), 246–256 (1979).
17. F.F. Yao, Speed-up in dynamic programming, *SIAM Journal on Algebraic Discrete Methods* **3**, 532–540 (1982).
18. D.E. Knuth, Optimum binary search trees, *Acta Informatica* **1**, 14–25 (1971).
19. T.C. Hu and M.L. Wachs, Binary search on a tape, *SIAM Journal on Computing* **16**, 573-590 (1987).
20. G.B. Dantzig, A.J. Hoffman and T.C. Hu, Triangulations (tilings) and certain block triangular matrices, *Mathematical Programming* **31** (1–14) (1985).
21. T.C. Hu and M.T. Shing, An optimum algorithm for matrix chain product, part I, *SIAM Journal on Computing* **11** (2), 362–373 (1982).
22. T.C. Hu and M.T. Shing, An optimum algorithm for matrix chain product, part II, *SIAM Journal on Computing* **13** (2), 228–251 (1984).

23. F.Y. Chin, An $O(n)$ algorithm for determining near-optimal computation order of matrix chain products., *Communications of the ACM* **21** (7), 544–549 (1978).
24. T.C. Hu and M.T. Shing, An $O(n)$ algorithm to find a near-optimum partition of a convex polygon, *Journal of Algorithms* **2**, 122–138 (1981).
25. M.T. Shing, Optimum ordered bi-weighted binary trees, *Information Processing Letters* **17** (2), 67–70 (1983).
26. R.K. Ahuja, T.L. Magnanti and J.B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, NJ, (1993).
27. R.E. Gomory and T.C. Hu, Multi-terminal network flows, *SIAM Journal on Applied Mathematics* **9** (4), 551–570 (1961).
28. L.R. Ford and D.R. Fulkerson, Maximal flow through a network, *Canadian Journal of Mathematics* **8**, 399–404 (1956).
29. L.R. Ford and D.R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, (1962).
30. D. Gusfield, Very simple methods for all pairs network flow analysis, *SIAM Journal on Computing* **19** (1), 143–155 (1990).
31. R. Hassin, Solution basis of multi-terminal cut problems, *Mathematics of Operation Research* **13** (4), 535–542 (1988).
32. C.K. Cheng and T.C. Hu, Maximum concurrent flow and minimum cut, *Algorithmica* **8**, 233–249 (1992).
33. T.C. Hu and E.S. Kuh, Editors, *VLSI Layout: Theory and Design*, IEEE Press, New York, (1985).
34. T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley & Sons, Chichester, West Sussex, England, (1990).
35. C.K. Cheng and T.C. Hu, Ancestor tree for arbitrary multi-terminal cut functions, In *Integer Programming and Combinatorial Optimization*, (Edited by R. Kannan and W.R. Pulleyblank), University of Waterloo Press (1990); Extended version: *Annals of Operations Research* **33**, pp. 199–213, (1991).
36. B. Korte, L. Lovász and R. Schrader, *Greedoids*, Springer-Verlag, Berlin, (1991).
37. D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation*, Prentice Hall, Englewood Cliffs, NJ, (1989).
38. P.G. Bradford, Efficient parallel dynamic programming, Technical Report 352, Department of Computer Science, Indiana University, (April 1992).
39. P. Ramanan, An efficient algorithm for finding an optimal order of computing a matrix chain product, Technical Report WSUCS-92-2, Department of Computer Science, Wichita State University, (1992).