Title	An Extended Depth-first Search : How to Decrease Backtracking (New Developments of Theory of Computation and Algorithms)
Author(s)	Kiniwa, Jun
Citation	数理解析研究所講究録 (2001), 1205: 1-6
Issue Date	2001-05
URL	http://hdl.handle.net/2433/40991
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

数理解析研究所講究録 1205 巻 2001 年 1-6

An Extended Depth-first Search — How to Decrease Backtracking —

神戸商科大学　木庭 淳 (Jun Kiniwa)

Department of Management Science, Kobe University of Commerce,
8-2-1 Gakuen nishi-machi, Nishi-ku, Kobe-shi, 651-2197 Japan
kiniwa@kobeuc.ac.jp

1 Introduction

In usual algorithms based on the depth-first search, there is no problem about this method. The number of steps for this search is $\Theta(V+E)$. So most people have paid no attention to this method. However, there are some areas, say distributed algorithms, in which one step costs much time. In particular, the backtracking property is a drawback in distributed mutual exclusion because no process is allowed to get into the critical section during the time. For an exhaustive search in the Web, the backtracking cost is extremely high. In this case, it is difficult to know the whole link structure in advance. Thus we cannot take an approach such as examining a near-hamiltonian circuit. So it is worth while to investigate how to decrease backtracking.

Since Tarjan's research [8] on the depth-first search, its technique has been widely used in computer science. Though other variant (parallel) traversal algorithms, K-depth or breadth-depth search, are also known [7], they are not so popular. From the viewpoint of artificial intelligence, Freuder [3, 4] investigated the condition of backtrack-free or backtrack-bounded structure. In the context of distributed algorithms, the depth-first search technique is also used [5]. It seems, however, that there are few efforts to reduce backtracking. So our research may be a fundamental work on these areas.

In this paper we present an extended depth-first search. The traversal does not always move along the tree if possible. The characteristics of our method are :

- If all the reachable nodes have been visited along the traversal starting from a node u to a node v, we can return directly from v to u.
- If the underlying graph has articulation points, our method is effective.

The rest of this paper is organized as follows. Section 2 presents our basic method and proves its correctness. It also involves successful probabilities in some graphs. Section 3 introduces an additional way which makes our method more effective. Section 4 concludes the paper.

2 Basic Method

2.1 Algorithm

In this section we describe our algorithm. Fundamental terminologies concerning graph theory can be seen in [9]. Since our idea reminds us of flow problems, we use a word "*flow*" to express some quantity. Suppose that the numbers of nodes and edges are unknown. Informally, our method works as follows. For each unvisited adjacent node, 1 flow is distributed so that the nodes to be visited can be memorized. The distributed flow is collected when the search visits the node. This is because there is no need to memorize the visited nodes any more. In this way, the distribution and the collection of flow is iterated repeatedly. Gradually the collected flow grows, meaning that the number of visited nodes grows. At the end, if we can collect the same amount of the initial flow, it means that every node has been visited. If we are at the node adjacent to the input node then, we can return without backtracking.

Let $flow(v)$ be the flow that the node v currently possesses, and $level(v)$ the amount of flow when the node v was visited. Let us consider the portion of a search from the starting node to the output node. Suppose that the length of the search from the starting node, $order(v)$, is associated with each node v. When a flow is distributed to a node v from adjacent nodes $\{u\}$, the minimum order of the received flow

$$minorder(v) = \min_{(u,v)\in E}\{order(u) \mid v \in S(in, out)\}$$

is recorded. When a node v is visited, it distributes a flow for each unvisited adjacent node. Then the maximum order of the adjacent nodes

$$maxorder(v) = \max_{(v,u)\in E}\{order(u) \mid v \in S(in, out)\}$$

is recorded. Note that the order is not defined when we are at v_{out} if the node has not been visited yet. Its order, however, is greater than $order(v_{out})$. We introduce the ***circuit condition*** which enables us to make a circuit.

Definition 1. *We say that an output node v_{out} satisfies the circuit condition for an input node v_{in} if, for every node $v \in S(in, out)$,*

$$order(v_{in}) \leq minorder(v), \tag{1}$$

$$maxorder(v) \leq order(v_{out}), \quad \textit{and} \tag{2}$$

$$level(v_{in}) = flow(v_{out}). \tag{3}$$

□

In particular, the first two inequalities are called the order conditions, where the expression (1) (resp. (2)) guarantees there is no inflow (resp. outflow) from $S(start, in)$ to $S(in, out)$ (resp. from $S(in, out)$ to $S(out, term)$). The third expression is called the flow condition, which guarantees the flow conservation. We just refer to the circuit condition if we mean all of them.

Without loss of generality, suppose that a connected graph is given.

Algorithm ExtendedDFS

input: a graph $G = (V, E)$ and a sufficient $flow(v_{start})$ for the starting node $v_{start} \in V$
output: a sequence of nodes with no more backtrackings than the depth-first search

ExtendedDFS(v)
begin
  Receive a flow into $flow(v)$ from the previous node ;
  if v is visited **then**
    Subtract (from $flow(v)$) and distribute 1 flow
    to each unvisited adjacent node ;
  if there is some unvisited adjacent node u **then**
    ExtendedDFS(u) ;
  else if v is not the input node v_{in} **then begin**
    if an adjacent node w satisfies the circuit condition **then**
      ExtendedDFS(w) ;
    else *ExtendedDFS*(p) for v's parent p ;
  end
end.

In usual depth-first search, if a node has more than one unvisited adjacent nodes, an arbitrary node is selected. The example below illustrates the behavior of our algorithm and the disadvantage of such nondeterminism which will be investigated in Section 2.3.

Example 1. Consider the scenario depicted in Fig. 1. When we visit (the gray node) v_{in}, we first distribute 1 flow for each adjacent node (a dotted arrow in (a)). Then we explore an unvisited node. In this case we have three choices for v_1, v_2 and v_3, and we select v_2 (a solid arrow in (b)). Next we iterate distributing 1 flow (c), exploring v_3 (d), and fail to make a circuit because $flow(v_3) = 5$ and $level(v_{in}) = 6$. We have to go backward just like a depth-first search (e)–(f). If we selected v_1 in (b), we could successfully make a circuit. □

Example 2. Fig. 2 shows the structure of search which enables us to make a circuit. The bold arrows mean the routing of search. Since there are no direct edges between $S(in, out)$ and other portions, the flow is conserved. □

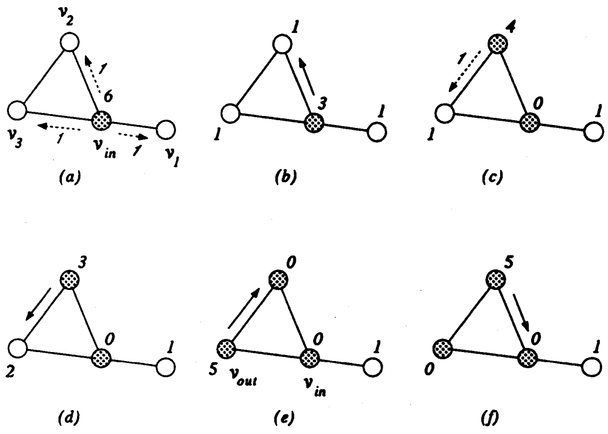

Fig. 1. Behavior of *ExtendedDFS*(v)

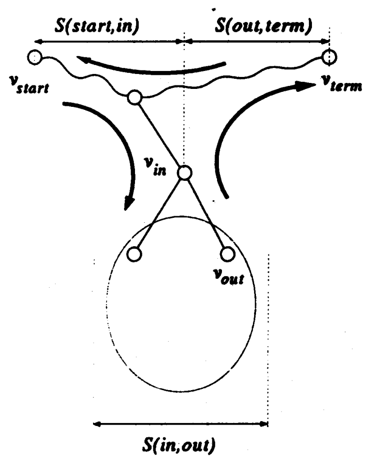

Fig. 2. Structure of search

2.2 Correctness

In this section we show the correctness of our *ExtendedDFS*(v). The following lemma states the relation between the order conditions and the flow conservation.

Lemma 1. *If an output node v_{out} satisfies the order conditions for an input node v_{in}, the flow conservation holds between v_{in} and v_{out}.*

Proof. Since our algorithm distributes a flow for each adjacent node when we visit a node v, the condition (1) means that there are no edges between the nodes on $S(start, in)$ and those on $S(in, out)$. Similarly, the condition (2) means that there are no edges between the nodes on $S(in, out)$ and those on $S(out, term)$. Thus the flow conservation must be held during the search $S(in, out)$. □

Lemma 2. *Suppose that the order condition is satisfied. The output node v_{out} satisfies the flow condition for the input node v_{in} if and only if all the nodes in the component have been visited.*

Proof. Since the order condition holds, the flow is conserved. Suppose that an output node v_{out} satisfies the flow condition for an input node v_{in}, and that there is some unvisited node adjacent to some node $u \in S(in, out)$. Then such node u has not received a flow because the output node collects the entire flow. Hence the adjacent nodes to u also have not. This is because u would have distributed if they had received. Since the component is connected, all the nodes have not been visited; a contradiction.

Next suppose that all the nodes in the component have been visited. Since no flow is distributed to visited nodes, each node can collect all the flow to be received. Thus all the flow input at the input node is collected by the output node. □

2.3 Properties

Now we examine probabilities of making a circuit in some graphs. For simplicity, the distribution of flows is not explicitly described here. We call a node of degree at least three a *branch node.*

Theorem 1. *If there exist disjoint paths from any branch node to the output node, the search always makes a circuit.*

Proof. Consider a branch node v with unvisited adjacent nodes $U = \{v_1, \ldots v_d\}$. Suppose that every node reachable from ancestors of v has been visited. Then we visit v and apply our *ExtendedDFS*(v) to U from v_1 to v_d. Let v_d be the last visited node in U (other nodes $U - v_d$ are visited in a depth-first manner). Then all the reachable nodes from $U - v_d$ are visited. Even if v_{out} is visited before other nodes, there exist disjoint paths from v_{out} to them. Thus we can go forward from v_{out} and backtrack to v_{out} repeatedly as shown in Fig. 3. If there is no branch node in the descendants of v_d, then we can make a circuit because every node can be explored and there is a disjoint path from v_d to the output node. Otherwise, we have unvisited v_d and some branch nodes in its descendants. Thus we can show the fact by induction. □

Corollary 1. *If the given graph is a complete graph or a ring, the search always makes a circuit.*

Proof. Both complete graphs and rings satisfy the condition of Theorem 1. □

In what follows, we show how to compute the probability of making a circuit in some series-parallel graphs. In usual, a series-parallel graph is defined to be a multigraph. We, however, only consider a simple graph and slightly modify the definition in [2].

Definition 2. *[2] A simple series-parallel graph (SP) is a triple $SP = (G, s, t)$, where $G = (V, E), V \supset V' \neq \phi$ and $V - V' = \{s, t\}$, of a simple graph with the source s of degree one and the sink t of degree one. Some edges (s, s') and (t', t) for $s', t' \in V'$ are incident on s and t, respectively. The series composition of SPs $((V_1, E_1), s_1, t_1)$ and $((V_2, E_2), s_2, t_2)$ with $t_1 = s_2' \in V_2'$ or $s_2 = t_1' \in V_1'$ is the SP $((V_1 \cup V_2, E_1 \cup E_2), s_1, t_2)$. The parallel composition of SPs $((V_1, E_1), s_1, t_1)$ and $((V_2, E_2), s_2, t_2)$ with $(s_1' = s_2) \wedge (t_1' = t_2)$ for $s_1' \neq t_1'$ is the SP $((V_1 \cup V_2, E_1 \cup E_2), s_1, t_1)$, called a parallel composition of the first kind, and with $(s_1 = s_2) \wedge (t_1 = t_2)$ is the SP $((V_1 \cup V_2 \cup \overline{V}, E_1 \cup E_2 \cup \overline{E}), \bar{s}, \bar{t})$, where $\{s_1, s_2, t_1, t_2\} \in V' = V_1 \cup V_2$, $\overline{V} = \{\bar{s}, \bar{t}\}$ and $\overline{E} = \{(\bar{s}, s_1), (t_1, \bar{t})\}$, called a parallel composition of the second kind.* □

Informally, we just append two edges whose ends are s and t if no one-degree terminals are generated. If we regard s as v_{in} and t as v_{out}, the probability $p(SP)$ of making a circuit in SP can be obtained on condition that each unexplored edge is selected equally likely. The most fundamental SP consisting of an *intermediate node* u is $C = (\{s, u, t\}, \{(s, u), (u, t)\}, s, t)$ with $p(C) = 1$, called a *series-parallel unit.*

Lemma 3. *The composition of k series-parallel units $C_1, \ldots, C_k$ generates the composed probability $\prod_{i=1}^{k} p(C_i) = 1$ if it is the series composition, and $\frac{1}{k^2} \sum_{i=1}^{k} p(C_i) = \frac{1}{k}$ if it is the parallel composition of the second kind* [1]*.*

Proof. It is obvious for the series composition. Next consider the parallel composition. Suppose we select one of parallel edges from $s' \in V'$ with probability $\frac{1}{k}$ and reach $t' \in V'$ via an intermediate node. At the node t', there are k unexplored edges incident on t' and we have to select the one other than $(t', \bar{t})$ for $\bar{t} \in \overline{V}$ to make a circuit (with probability $\frac{k-1}{k}$). After selecting the one, we have to backtrack from the intermediate node because the node s' has already been visited. At the node t' again, we have to select one other than $(t', \bar{t})$ from $k - 1$ unexplored edges, and so forth. In this way, the probability of visiting every node before $\bar{t} \in \overline{V}$ is $\frac{k-1}{k} \frac{k-2}{k-1} \cdots \frac{1}{2} = \frac{1}{k}$. Since there are k choices for the edge from s', we have $k \cdot \frac{1}{k} \cdot \frac{1}{k} = \frac{1}{k}$. Fig. 4 (a)—(c) illustrate the case for $k = 3$. □

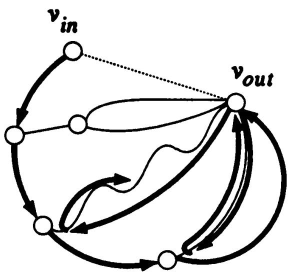

Fig. 3. Disjoint paths from every branch node

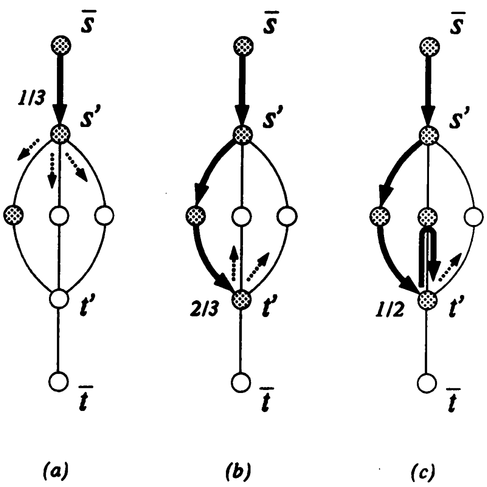

Fig. 4. Composed probability for series-parallel units

[1] Note that the parallel composition of the first kind cannot be defined for series-parallel units.

Lemma 4. *Let $SP_1, \ldots, SP_k$ be simple series-parallel graphs. The composition of $SP_1, \ldots, SP_k$ generates the composed probability $\prod_{i=1}^{k} p(SP_i)$ if it is the series composition, and $\frac{1}{k^2}\sum_{i=1}^{k} p(SP_i)$ if it is the parallel composition.*

Proof. Consider the case for $k = 2$. If we have a series composition, the composed probability is $p(SP_1) \cdot p(SP_2)$ because every node in SP_1 must be successfully visited before the input node of SP_2. If we have a parallel composition, there are two choices from $s' \in V'$. Suppose that we first select SP_1 with probability $\frac{1}{2}$. After visiting every node in SP_1, we reach t with probability $p(SP_1)$. Then there are two choices at $t' \in V'$ and select SP_2 with probability $\frac{1}{2}$. Every node in SP_2 is visited with probability 1 because backtracking begins until arriving at t'. Then we traverse $(t', \bar{t})$ and make a circuit. If we first select SP_2, the graph is similarly traversed as above. Thus the probability is $\frac{1}{2} \cdot (p(SP_1) + p(SP_2)) \cdot \frac{1}{2}$.

Next suppose that our claim holds for $k - 1$. For series composition, it is easily verified that the composed probability is $(\prod_{i=1}^{k-1} p(SP_i)) \cdot p(SP_k)$ because the traversal cannot successfully proceed until the first $k-1$ composed graph is traversed. For parallel composition, let SP_{k-1}^{1} denote the graph applying the parallel composition to $SP_1, \ldots, SP_{k-1}$. If we first select SP_k from the node s' with probability $\frac{1}{k}$, we can successfully reach t' with probability $p(SP_k) \cdot (\frac{k-1}{k}\frac{k-2}{k-1} \cdots \frac{1}{2})$. If we first select SP_{k-1}^{1} from the node s' with probability $\frac{k-1}{k}$, the successful probability of traversal is obtained by conditioning the first choice from t', that is $\frac{k-1}{k}$. Then the successful probability of traversal is $\frac{1}{(k-1)^2}\sum_i p(SP_i)$ by the induction hypothesis. Thus we have $\frac{1}{k} \cdot p(SP_k) \cdot (\frac{k-1}{k}\frac{k-2}{k-1} \cdots \frac{1}{2}) + \frac{k-1}{k} \cdot (\frac{1}{(k-1)^2}\sum_{i=1}^{k-1} p(SP_i)) \cdot \frac{k-1}{k} = \frac{1}{k^2}\sum_{i=1}^{k} p(SP_i)$. □

Theorem 2. *If we have a simple series-parallel graph by disconnecting an edge (v_{in}, v_{out}) of a given graph, the probability of making a circuit can be obtained by the same operations of which it is composed.* □

3 Labeling Method

This section provides a useful method which enables us to make a circuit. It is effective if we know the entire graph structure and we can specify cycles. Thus, unlike the previous section, we label the nodes in advance and traverse the graph. Our idea is to delay visiting the output node until all the nodes in the component are visited. For this purpose, we find biconnected components using depth-first search, and label the nodes consistently with the distance from the output node. Then whenever we come to a branch node, we first select the largest labeled node. Our algorithms have properties as follows.

- For any connected graph with at least one cycle, we can make a circuit.
- If it has k biconnected components, we can make circuits k times.

DesignBC

input: a connected graph $G = (V, E)$ containing at least one cycle
output: a node-labeled, edge-marked graph which enables us to make a circuit

begin

Find biconnected components $BC = \{bc_1, bc_2, \ldots, bc_k\}$, where bc_i $(i \geq 2)$ shares an articulation point with some bc_j $(j < i)$.
for $i := 1, \ldots, k$ **do begin**

1. Let the input node $v_{in} \in bc_i$ be the articulation point shared with bc_j $(j < i)$, and $v_{out} \in bc_i$ the output node adjacent to v_{in}.
2. Disconnect the edge (v_{in}, v_{out}).
3. For bc_i, construct a spanning tree rooted at v_{out} with a leaf v_{in}, and mark the tree edges.
4. Label the nodes of spanning tree the distance from v_{out} [2].

[2] The labeled v_{in} in the previous iteration is not relabeled again.

Our *ExtendedDFS*(v) is changed regarding how to distribute a flow, which node to be selected, and how to traverse.

LabelExtendedDFS

- When we visit an articulation point $v_{in} \in bc_i$, a flow is not distributed outside the component bc_i.
- At a branch node of the spanning tree, we next select the unvisited largest labeled node.
- We traverse the spanning tree (only marked tree edge) from v_{in} to v_{out}.

For the first rule, if a flow is distributed outside the biconnected component, the output node cannot satisfy the flow condition. If the given graph has an articulation point with a bridge as shown in Fig. 2, there is no need to keep this rule. The second rule is related to our idea described above. The third rule forces the search to proceed along the path.

Lemma 5. *If the path of $S(in, out)$ is a biconnected component with an articulation point v_{in}, the output node v_{out} satisfies the circuit condition for the input node v_{in} by our LabelExtendedDFS(v).*

Proof. Since the biconnected component can be reached only via an articulation point, the order conditions are satisfied. Let v_b be any node with more than two degrees in a spanning tree. Since the children of v_b have larger label than its parent, our *LabelExtendedDFS*(v) visits all the children before the parent. Thus every node in the spanning tree has been visited when the search comes to v_{out}. Thus the flow condition is satisfied from Lemma 2. □

Definition 3. *[6] A biconnected tree $T_G = (V_t, E_t)$ of a graph G is a tree whose node $v \in V_t$ corresponds to a biconnected component bc_v in G, and T_G has an edge $(u, v) \in E_t$ if two biconnected components bc_u and bc_v share an articulation point.* □

Lemma 6. *If we regard the graph G as a biconnected tree T_G, then the LabelExtendedDFS visits the nodes $v \in V_t$ in a depth-first manner.* □

From lemmas above, the next theorem holds.

Theorem 3. *Given a graph with k biconnected components, DesignBC can order the nodes such that LabelExtendedDFS makes circuits k times.* □

4 Conclusion

We investigated how to decrease backtracking. The circuit condition guarantees every node has been visited. Thus we can make a circuit if the condition holds. The condition has also revealed a structure of graphs, biconnected components, which enables us to make circuits. This work has an application to the problems in which backtracking cost is extremely high.

References

1. E.M.Bakker and J.Van Leeuwen, "Uniform d-emulations of rings, with application to distributed virtual ring construction," *Networks*, 23: 237–248, 1993.
2. A.Brandstädt, V.B.Le and J.P.Spinrad, "Graph classes: a survey," *SIAM Monographs on Discrete Mathematics and Applications*, SIAM, 1999.
3. E.C.Freuder, "A sufficient condition for backtrack-free search," *Journal of the Association for Computing Machinery*, 29, 1: 301–305, 1982.
4. E.C.Freuder, "A sufficient condition for backtrack-bounded search," *Journal of the Association for Computing Machinery*, 32, 4: 755–761, 1985.
5. S.A.M.Makki and G.Havas, "Distributed algorithms for depth-first search," *Information Processing Letters*, 60: 7–12, 1996.
6. U.Manber, "Introduction to algorithms : a creative approach," *Addison-Wesley*, 1989.
7. E.Reghbati and D.G.Corneil, "Parallel computations in graph theory," *SIAM Journal on Computing*, 7, 2: 230–237, 1978.
8. R.E.Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, 1, 2: 146–160, 1972.
9. D.B.West, "Introduction to graph theory," *Prentice-Hall*, 1996.