CSOR W4231.002 Lecture 8: Mar 31, 2014 Spring 2014

Eleni Drinea

In this lecture we discuss Depth-First-Search (our second graph traversal algorithm), and applications, including topological sorting on a directed acyclic graph and finding strongly connected components in a directed graph.

1 Depth-First-Search (DFS)

A second natural method for solving s-t connectivity is the approach you might use if you were in a maze and wanted to find your way out (think of vertices as interconnected rooms and of edges as corridors between different rooms).

1. Start at s, and try the first edge out of s, towards some node v.
2. Continue from v until you reach a "dead end", that is, a node whose neighbors have all been explored.
3. **Backtrack** to the first node with an unexplored neighbor and repeat 2.

This procedure is called Depth-First-Search (DFS): it explores as deeply as possible and then backtracks. DFS naturally defines a tree, called the DFS tree: every time an "unexplored" vertex v is discovered, it is added as a child of the node u that is responsible for "discovering" it, via an edge (u, v).

DFS uses a stack to keep the explored vertices. The stack is implicit in the following pseudocode. Also, procedures previsit(u) and postvisit(u) are used to keep information relevant to the application we want to use the DFS traversal of the graph for. For example, they might record the time DFS starts exploring vertex u (thus u is pushed in the stack) and the time all of its neighbors have been explored (so u is popped from the stack and is never pushed into the stack again). Also, they might record which node is responsible for discovering u, that is, the parent of u in the DFS tree.

```
DFS( G(V,E) )
  for u ∈ V do
     explored[u] =0
  end for
  for u ∈ V do
     if explored[u] == 0 then Search(u)
     end if
  end for

Search(u)
  previsit(u)
  explored[u] = 1
  for (u,v) ∈ E do
     if explored[v] == 0 then Search(v)
     end if
  end for
  postvisit(u)
```

The running time of the algorithm is $O(n+m)$ if previsit() and postvisit() require $O(1)$ time: Search(u) is called once for every vertex, and each call to Search(u) requires $O(1)+deg(u)$ time, since the conditional statement in Search() statement is executed once for every edge in the adjacency list of u.

1.1 Similarities and differences between BFS and DFS

Unlike BFS(G, s), we implement DFS(G) so that it explores *all* the vertices in the graph. That is, if the first vertex of the graph to be explored is u, then Search(u) returns after exploring the connected component of u since there is a path from u to every vertex in the connected component; or, in directed graphs, after exploring all vertices reachable from u. If there are still unexplored vertices in the graph, then DFS re-starts from one of them. Since Search(u) produces a tree rooted at u, DFS(G) produces a *forest* of trees; if the graph is undirected, these trees correspond to the connected components of the graph.

Similarities between BFS and DFS:

- Both procedures compute the connected component of the vertex they start exploring from (or the set of nodes reachable from that vertex).
- Both have the same efficiency (linear time).

Differences between BFS and DFS:

- Order of visiting the vertices: when BFS discovers a vertex, it defers exploring until all vertices in the same layer have been discovered. When DFS discovers an unexplored vertex, it moves on to exploring it right away. Thus DFS moves along long paths, potentially very far from s, before backtracking to a vertex with unexplored neighbors.

1.2 Examples

Consider the undirected graph in Figure 1(a). Let A_0 be the adjacency list representation for G where the vertices in every adjacency list are ordered by increasing index. Then DFS on input A_0 yields the tree in Figure 1(b).

Similarly to BFS, the order in which the vertices appear in the adjacency lists matters for the tree produced by DFS. For example, consider the alternative representation for G below, denoted by A_1. Here $N(i)$ denotes the (ordered) adjacency list of vertex i. On input A_1, DFS yields the tree in Figure 1(c).

$$\begin{aligned} A_1 \;=\; & \{N(1)=\{3,2\}, N(2)=\{1,4,5,6\}, N(3)=\{1,7,6,8\}, \\ & N(4)=\{2,5\}, N(5)=\{2,4,6\}, N(6)=\{3,2,5\}, N(7)=\{8,3\}\} \end{aligned}$$

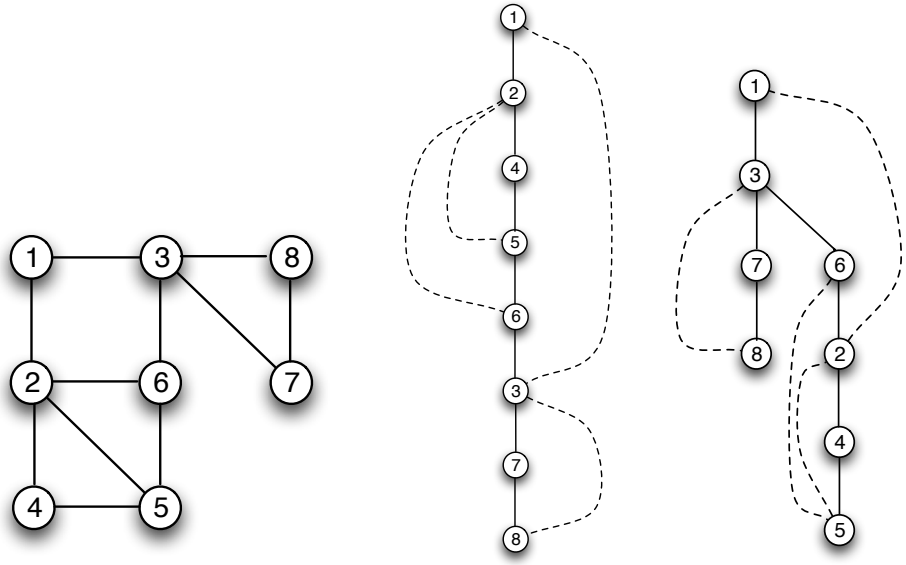
Now consider the directed graph G in Figure 2(a). Assuming an adjacency list representation where the vertices in every adjacency list are ordered by increasing index, DFS(G) yields the forest in Figure 2(b).

1.3 Classification of graph edges

We will now classify the graph edges that do not belong to the DFS tree. This will be useful for deducing structural properties of the graph as we will soon see.

We say that a vertex u is an ancestor of a vertex v if it appears on the path from the root of the DFS to v. Then v is a descendant of u. Note that this ancestral relationship translates to undirected graphs as well: since the tree is rooted, if two nodes are on the same path from the root, the node at largest depth is the descendant of the other.

We first consider directed graphs.



(a) Graph G (b) DFS on input A_0 (c) DFS on input A_1

Figure 1: Two DFS trees for the graph in Figure 1(a), given adjacency list representations A_0 and A_1 respectively. Solid edges belong to the trees while dashed edges are graph edges but not tree edges.

1. **Forward edges** go from an ancestor to a descendant other than a child; e.g., (u, v) in Figure 3(a).
2. **Back edges** go from a descendant to an ancestor, other than the parent; e.g., (v_3, v_1) in Figure 2(b).
3. **Cross edges** go from right to left (no ancestral relation), that is,
   - from tree to tree; e.g., (v_5, v_4) in Figure 2(b).
   - between nodes in the same tree but in different branches (no ancestral relationship); e.g., (u, v) on the right of Figure 3(b).

In undirected graphs we may only have **back** and **tree** edges: cross edges would have appeared as tree edges and forward edges would have appeared as back edges.

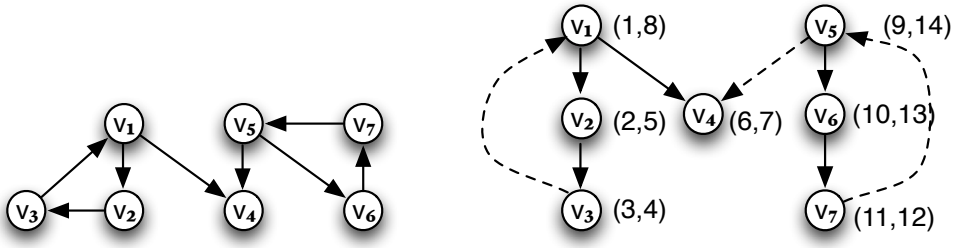
1.4 Properties of DFS: Time

As already mentioned, one natural use of previsit() and postvisit() is to keep a counter that is incremented by 1 every time one of these routines is accessed. This naturally corresponds to a notion of time. Thus we can maintain the pair $(start(u), finish(u))$ for all $u \in V$. Figure 2(b) shows these pairs for every vertex in G.

If we use an explicit stack, then $start(u)$ corresponds to the first time u is pushed in the stack and $finish(u)$ corresponds to the time that u is popped from the stack (that is, all its neighbors have been explored and it will never be visited again).

Fact 1 *For two vertices u, v, the intervals $[start(u), finish(u)]$ and $[start(v), finish(v)]$ either are disjoint or one contains the other.*

If there is an ancestral relationship between u and v, then one of the two intervals contains the other. If u, v appear in different trees or different branches of the same tree, their intervals are disjoint.



(a) Directed graph G (b) DFS forest

Figure 2: The construction of the DFS forest for the graph in Figure 2(a). Solid edges belong to the trees in the DFS forest while dashed edges are graph edges but not tree edges.

Fact 2 *At any time, if s is the vertex at the bottom of the stack and v is the vertex at the top of the stack, the contents of the stack form a path from s to v in the graph.*

If u is the last vertex in the stack before v, then v was pushed in the stack because there is an edge from u to v. The same argument applies to every vertex before u, hence we can trace back these vertices to get a path from the vertex s at the bottom of the stack to the vertex v at the top of the stack.

Claim 1 *For any edge $(u, v) \in E$, we have*

$$start(v) < start(u) < finish(u) < finish(v)$$

if and only if (u, v) is a back edge in the DFS tree.

Proof. If (u, v) is a back edge, v is an ancestor of u, thus the interval of v contains the interval of u by Fact 1. Otherwise, if $start(v) < start(u) < finish(u) < finish(v)$, then v was pushed in the stack before u and is still in the stack when u is pushed into it. By Fact 2, v is on the path from the root to u in the DFS tree. Then (u, v) is a back edge. □

Similarly, we can identify conditions for the intervals of the two endpoints of an edge (u, v) for the other types of graph edges (see Figures 3(a) and 3(b)).

Fact 3 *A graph edge (u, v) is a forward edge in the DFS tree if and only if*

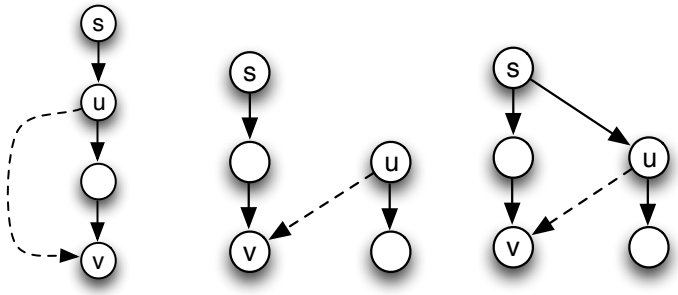
$$start(u) < start(v) < finish(v) < finish(u).$$

Fact 4 *A graph edge (u, v) is a cross edge in the DFS tree if and only if*

$$start(u) > finish(v).$$

2 Applications

We are now ready to discuss applications of DFS.



(a) Forward edge (b) Cross edge between different trees and within the same tree

Figure 3: Forward and cross edges.

2.1 Detection of cycles

Claim 2 *A graph $G(V, E)$ has a cycle if and only if the DFS of $G(V, E)$ yields a back edge.*

Proof. If (u, v) is a back edge, together with the path on the DFS tree from v to u, it forms a cycle.

Conversely, suppose G has a cycle. Run DFS on G. Let v be the first vertex from the cycle discovered by DFS. Let (u, v) be the preceding edge in the cycle. Since there is a path from v to every vertex in the cycle, all vertices in the cycle will now be discovered and appear in the subtree rooted at v (*why?*).

In particular u will be discovered, and appear somewhere in the subtree rooted at v. Thus u is a descendant of v and, by Fact 1, the interval of u is contained within the interval of v, that is, $start(v) < start(u) < finish(u) < finish(v)$ (since $start(v) < start(u)$ by assumption). By Claim 1, the edge (u, v) is a back edge. □

2.2 Topological Sorting

2.2.1 Directed Acyclic Graphs (DAGs)

If an undirected graph has no cycles, then it has an extremely simple structure: it is a tree. Thus it can have at most $O(n)$ edges.

However for a directed graph, it is possible to have no cycles yet still be dense. For example, consider the graph on vertex set $\{1, 2, \ldots, n\}$ and edges $(i, j) \in E$ if $i < j$. Such a graph has $\binom{n}{2}$ edges and no cycles.

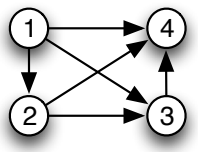


Figure 4: A DAG on 4 nodes and $\binom{4}{2}$ edges.

Directed acyclic graphs are so important in computer science that they have their own acronym, naturally enough, DAGs.

2.2.2 Topological Ordering via properties of DAGs

Consider a set of tasks labeled $\{1, 2, \ldots, n\}$ that need to be performed and a set of dependencies for certain pairs i and j indicating that task i must be performed before task j. For example, tasks may correspond to courses and certain courses are prerequisites for others.

Given a set of tasks and a set of dependencies (precedence relations), we want a valid order in which the tasks could be performed, so that all dependencies are respected.

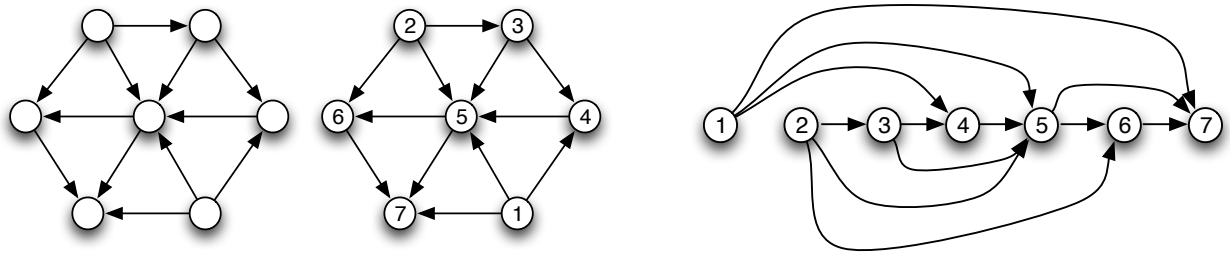
We can naturally model this problem using a directed graph where tasks are vertices and a **directed** edge (i, j) indicates that task i must be performed before task j.

Also, for the precedence relations to be meaningful, the resulting graph must be a DAG. For assume there is a cycle in G: since no task can begin until some other completes, no task in the cycle will ever be done, because no task can be done first.

Definition 1 *A topological ordering of G is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ such that for every edge (v_i, v_j), we have $i < j$.*

It follows that

- All edges point **forward** in the topological ordering (or sorting).
- The topological ordering provides an order in which all tasks can be safely performed: when we try to perform task j, all the tasks required to precede it have already been done. See Figure 5(b) for an example.



(a) A DAG and its topological order

(b) Topologically sorted DAG: edges go from left to right

Figure 5: Figure 5(a) shows a DAG and a topological order for it. Figure 5(b) presents a different drawing of the topologically sorted DAG that emphasizes the topological ordering.

Claim 3 *If G has a topological ordering, then G is a DAG.*

It is easy to show this claim by contradiction. Note that Figure 5(b) provides an immediate visualization of Claim 3: all edges go from left to right so there can be no cycles. Hence if a graph has a topological sorting, then the graph is a DAG.

Is the converse of Claim 3 true? That is, *does every DAG have a topological ordering?* And if yes, how do we find it?

To answer this, we will look closer at the structural properties of a DAG.

Fact 5 *At least one vertex in every DAG has no outgoing edges (*sink *vertex).*

Proof. Suppose there is no such vertex, that is, every vertex in the DAG has an outgoing edge. Start at a vertex u_1 and follow an outgoing edge to some u_2. For $i \geq 2$, we keep following the outgoing edge of u_i to vertex u_{i+1} (such an edge exists by assumption): then u_{i+1} is either one of the vertices visited so far in the path $u_1, \ldots, u_i$ (hence we have a cycle), or is a yet unvisited vertex. Since there are only n vertices in the graph, at the latest the outgoing edge from u_n forms a cycle. □

Fact 6 *At least one vertex in every DAG has no incoming edges (*source *vertex).*

Proof. The proof is identical to the previous one, the only difference being that we now follow the path of incoming edges in the reverse direction. □

Hence we proved the following fact.

Fact 7 *Every DAG has at least one source and at least one sink.*

We may now use this fact to find topological order: the key observation is that the node that we put first in the topological order must have no incoming edges, so that it corresponds to a task that may be performed first without violating any precedence constraints. Fact 7 guarantees that such a task exists: it is a source vertex in the DAG. So we may label any source as first in the topological order, and remove it and its outgoing edges from the graph. The remaining graph G' is a DAG: removing edges from G cannot yield a cycle! Therefore G' has at least one source. This motivates an inductive proof for showing that all DAGs have a topological order (*exercise*) and the following algorithm for finding one.

TopologicalSort(DAG G)

1. Find a source vertex u and order it first.
2. Delete u and its outgoing edges from G; let G' be the new graph.
3. TopologicalSort(G')
4. Append the order found after u.

The running time of this algorithm is $O(n^2)$ since TopologicalSort is called at most n times and each recursive call requires $O(n)$ time. It can be improved to $O(m+n)$ (*how?*).

2.3 Topological Sorting via DFS

The following algorithm provides another way to perform topological sorting.

- Given a DAG G, run DFS(G). ($O(n+m)$ time)
- Process the tasks in **decreasing** order of $finish$ times.

We claim that this gives a valid topological ordering. Before we give a formal proof, we give some intuition as to why this works. Consider the vertex (task) u with the largest $finish$ time. This vertex cannot have any incoming edges from any other vertex (equivalently, this task does not depend on any other task): if it did, that other vertex would have larger $finish$ time! So this task may safely be performed first.

Now consider the vertex v with the second largest $finish$ time: v cannot have incoming edges from any other vertex except (potentially) for u. Otherwise, the same argument as above shows that there would be another task, different from both u and v, with larger $finish$ time than v which contradicts the fact that v has the second largest $finish$ time. Hence v may be safely performed second. And so on and so forth. The formal proof follows.

Proof. Since there are no cycles, by Claim 2, there are no back edges in the DFS forest. Thus if $(u, v) \in E$ then either it is (a) a forward edge and $start(v) > finish(u) > finish(v) > start(u)$ by Fact 3; or, (b) a cross edge and $finish(u) > start(u) > finish(v)$ by Fact 4.

In either case $finish(u) > finish(v)$.

Consider when task v is processed. The tasks with precedence constraints **into** task v, that is, the tasks that must be completed before v, are all the tasks u such that there is an edge (u, v) in G. By our observation above, all such tasks u have $finish(u) > finish(v)$.

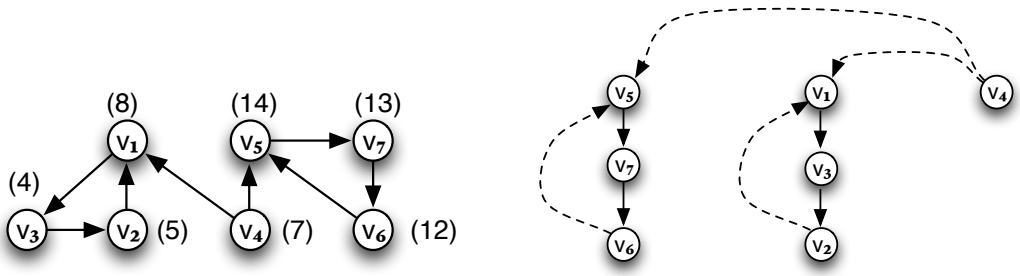
Since we are processing tasks in decreasing order of $finish$ times, all such tasks u have already been processed before v. □

2.4 Strongly connected components in directed graphs

A classic application of DFS is finding strongly connected components (SCCs) in directed graphs. Recall that an SCC is a set of nodes such that there is a path from u to v and from v to u for every pair u, v in the set. Decomposing a directed graph into its SCCs provides extremely useful information about the graph because it allows us to think of connectivity information of a directed graph in a hierarchical way: on a top level, we have a graph of SCCs which, as we shall soon see, has a useful and simple structure. If we want more details, we could look at a vertex of this graph and work with a completely connected subgraph. The following algorithm computes all SCCs of G in linear-time.

1. Run DFS(G); compute $finish(u)$ for all u.
2. Compute G^r.
3. Run DFS(G^r) in decreasing order of $finish(u)$.
4. Output the vertices of each tree in the DFS forest in step 3 as an SCC.

Consider again the example G in Figure 2(a). Its SCCs are given by the trees in Figure 6(b).



(a) G^r and $finish$ times from DFS(G)

(b) DFS(G^r) by decreasing $finish(u)$

Figure 6: In Figure 6(a), G^r and $finish$ times are with respect to G and DFS(G) in Figures 2(a) and 2(b) respectively. The SCCs of G are the trees in the forest in Figure 6(b).

Remark 1 *From now on, whenever we use start and finish times, they are the times obtained by running DFS(G) in step 1 of the algorithm above.*

The intuition behind this algorithm is similar to that in Section 2.3. However it is easier to think in terms of SCCs in this case, so we start with few easy observations about SCCs.

1. The strongly connected components of G and G^r are the same: if u is reachable from v in G and vice versa, then v is reachable from u in G^r and vice versa.
Consider the graph of all SCCs, that is, the graph obtained when we shrink every SCC into a vertex (a supervertex) and add an edge (a superedge) from SCC C_i to SCC C_j if there is an edge from some vertex in C_i to some vertex in C_j.
2. The graph of SCCs is a DAG. (See Figure 7(a) for a quick explanation.)

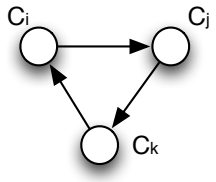


Figure 7: If SCCs C_i, C_j, C_k formed a cycle, they would be merged to a single SCC.

Now consider SCC $C_i = (V_i, E_i)$. We slightly extend our notation by defining

$$start(C_i) = \min_{v \in V_i} start(v), \quad finish(C_i) = \max_{v \in V_i} finish(v).$$

Lemma 1 *Let C_i and C_j be two strongly connected components in the directed graph G. Suppose there is an edge $(u, v) \in E$ where $u \in C_i$ and $v \in C_j$. Then $finish(C_i) > finish(C_j)$.*

Suppose you had proven this lemma. Then

Corollary 1 *Let C_i and C_j be two distinct SCC's in directed graph G. Suppose that there is an edge (u, v) in G^r such that $u \in C_i$ and $v \in C_j$. Then $finish(C_i) < finish(C_j)$.*

Proof. The SCCs are the same in G and G^r, so reverse the edge and apply Lemma 1 in G. □

We are now ready to argue correctness of our algorithm in an informal way. Let C_m be the SCC with the maximum $finish$ time among all SCCs. The vertex u with the maximum $finish(u)$ belongs to this SCC. Suppose we perform DFS in G^r starting with some vertex in C_m. The search will discover all vertices in this SCC, thus explore all of C_m. By Corollary 1, in G^r, there are no outgoing edges from C_m to any other SCC. Thus DFS will explore no other vertices and will output just C_m.

Now continue with the SCC with the second largest $finish$, say C_{m-1}. Again, DFS will only explore the vertices within this SCC: any outgoing edges from C_{m-1} may only be towards vertices in C_m, which has already been explored! So DFS will only output C_{m-1} before it gets stuck and re-starts from some unexplored vertex.

And so on and so forth. In particular, every time C_i is searched, its only outgoing edges will be towards SCC's that have already been explored. Therefore, when vertices in G^r are considered by decreasing $finish$ times, each DFS tree consists of exactly one SCC.

The formal proof of correctness for the algorithm is by induction on the number of SCCs (exercise).

Finally, we prove Lemma 1.

Proof. We will prove Lemma 1 by considering two cases.

1. $start(u) < start(v)$: DFS starts at C_i. Before Search(u) returns, edge (u, v) will be explored, thus v will be explored. Since every vertex in C_j is reachable from v and there is no edge from any vertex in C_j back to any vertex in C_i (*why? similarly, if C_j has edges towards some C_k, can C_k reach C_i?*), all of C_j will be explored before Search(v) returns. Thus v is assigned a $finish$ time before DFS backtracks to u. Thus $finish(v) < finish(u) \Rightarrow finish(C_j) < finish(C_i)$.

2. $start(u) > start(v)$: DFS explores C_j first. Since there is no edge from C_j to C_i, DFS will explore all of C_j (and potentially some other SCC's) before it re-starts from some unexplored vertex. In particular, all vertices of C_i will be unexplored (*why?*). Thus all vertices in C_j have smaller $finish$ times than the $start$ time of any vertex in C_i. Hence $finish(v) < start(u) < finish(u) \Rightarrow finish(C_j) < finish(C_i)$.

□