A COMBINATORIAL CHARACTERIZATION OF THE DISTRIBUTED 1-SOLVABLE TASKS [1]

Ofer Biran, Shlomo Moran and Shmuel Zaks

Department of Computer Science
Technion, Haifa, Israel 32000

Revised Version, October 1989

ABSTRACT

Fischer, Lynch and Paterson showed in a fundamental paper that achieving a distributed agreement is impossible in the presence of one faulty processor. This result was later extended by Moran and Wolfstahl who showed that it holds for any task with a connected input graph and a disconnected decision graph.

In this paper we extend that latter result, and in fact we set an exact borderline between solvable and unsolvable tasks, by giving a necessary and sufficient condition for a task to be 1-solvable (that is: solvable in the presence of one faulty processor). Our characterization is purely combinatorial, and involves only relations between the input graph and the output graph, defined by the given task. It provides easy proofs for the non-solvability of tasks, and also provides a universal protocol which solves any task which is found to be solvable by our condition.

Using the above characterization, we also derive a novel technique to prove lower bounds on the number of messages that must be sent due to processor failure; specifically, we provide a simple proof that for each fixed $N > 2$ there exist distributed tasks for N processors, that can be solved in the presence of a faulty processor, but any protocol that solves them must send arbitrarily many messages in the worst case.

---

[1] This research was supported in part by Technion V.P.R. Funds - Wellner Research Fund and Loewengart Research Fund, by the I. Goldberg Fund for Electronics Research, and by the Foundation for Research in Electronics, Computers and Communications, administrated by the Israel Academy of Sciences and Humanities. A preliminary version of this paper appeared in the proceedings of the 7th PODC, Toronto, Canada, August 1988.

1. INTRODUCTION

An asynchronous distributed network consists of a set of processors, connected by communication lines, through which they may have to communicate in order to accomplish a certain task; the time delay on the communication lines is finite, but unbounded and unpredictable.

In recent years a number of papers that investigate impossibility issues in distributed networks were published. Some of these impossibility results stem from symmetry or from lack of information (like not having distinct identities to the processors or not knowing the size of the network); the work of [ASW] is of this kind. Other impossibility results are due to processors failures that are either naive (e.g., crash failures, or failures of the fail-stop type) or malicious (e.g., Byzantine faults); the works [FLP, MW] and [LSP, FLM] are, respectively, of these two types.

In this paper we study the case when at most one processor is faulty, which means that all of its messages are not delivered from some point on (crash failure). It was shown in [FLP] that it is impossible to achieve a distributed consensus for this case. This result was extended in several directions. In [DDS] the features of asynchrony that yield the result of [FLP] and related results were analyzed. In [DLPSW] it was shown that approximate consensus, in which all processors must agree on values that are arbitrarily close to one another, is possible in the presence of a few faulty processors. In [ABDKPR] a few other problems were shown to be solvable in the presence of faulty processors. However, giving a precise characterization of the tasks that can be solved in the presence of t faulty processors remains an interesting open problem (partial results are given in [TKM]). In this paper we provide such a characterization for the case $t = 1$.

The first step towards the result in this paper was done in [MW], where it was shown that any distributed task satisfying a certain combinatorial property is not 1-solvable (i.e., is not solvable in the presence of one faulty processor). Informally, the input values and the output values of a given problem were described in [MW] by input and output graphs, where a vertex in the input [output] graph is a vector of input [output] values of the processors, and there is an edge connecting two vertices if and only if the corresponding vectors differ in exactly one entry. It was shown there that any distributed task whose input graph is connected and whose output graph is disconnected is not 1-solvable.

In this paper we extend the condition in [MW] to provide a complete characterization of the asynchronous distributed tasks that are 1-solvable. This characterization is given in a pure graph-theoretic formulation, in terms of the input and output graphs of these tasks, and the relations between them. A simple protocol that solves tasks satisfying this characterization is also given. This protocol uses a novel technique, in which every processor eventually decides on a vertex in the output graph. The set of vectors decided upon is included in a set of two adjacent vertices, which implies that the actual output vector is one of these two vertices. (Convergence on one vertex is impossible, since it can be shown to contradict the above result in [FLP].)

Using our characterization, the question of whether a given task is 1-solvable is reduced, in many cases, to the technical problem of determining certain properties of a given graph. We demonstrate this by extending some known impossibility results to their extremes. In a subsequent paper [BMZ1] we use this characterization to show that the problem of deciding whether a given distributed task is 1-solvable is NP-hard.

The pure combinatorial properties of our characterization provide a simple technique for proving lower bounds on the number of messages needed to solve distributed tasks in the presence of a faulty

processor. More specifically, we show that for any fixed $N \geq 3$ there is a 1-solvable task for N processors, such that for every arbitrarily large M, there is an input for this task such that any protocol that solves it sends in the worst case more than M messages on this input. Previous lower bound proofs in similar models required a rather involved use of an adversary ([Fe]). In a subsequent paper [BMZ2] this characterization is used to achieve further results concerning the communication complexity of general 1-solvable tasks.

The rest of this paper is organized as follows: In Section 2 we present definitions and notations, and notions like decision tasks, protocols and 1-solvability are discussed. In Section 3 we present two conditions (Theorems 1 and 2) that must be met by protocols that 1-solves a given task. These two conditions are then used in Section 4 to derive our main result (Theorem 3), which provides a complete characterization of tasks which are 1-solvable, and presents a universal protocol that 1-solves such tasks. In section 5 we modify our result for the case where the identities of the processors are not mutually known. We conclude in Section 6 where we present the lower bound mentioned above.

2. DEFINITIONS AND NOTATIONS

2.1 Asynchronous Systems

An *asynchronous distributed network* is composed of a set $V = \{ P_1, P_2, \ldots, P_N \}$ of N *processors* ($N \geq 3$), each having a unique *identity*. We assume that the identities of the processors are mutually known, and w.l.o.g. that the identity of P_i is i. Our results are applicable also to the model in which the identities are not mutually known (or absent, provided that the inputs are distincts). The outline of the modifications needed in the definitions and the proofs required for this model is given in Section 5. The processors are connected by *communication links*, and they communicate by exchanging messages along them. Messages arrive with no error in a finite but unbounded and unpredictable time; however, one of the processors might be faulty (the exact definition is given in the sequel), in which case messages might not have these properties. The faults discussed in this paper are crash failures [FLP].

A network of N processors is viewed as an undirected graph with N vertices, each representing a processor. It is implicitly assumed in our proofs that the network is complete, but the results easily generalize to arbitrary biconnected networks, in which a failure of a processor cannot disconnect the network.

2.2 Decision Tasks

We view a decision task as a mapping of possible inputs to allowable outputs. For this we need few definitions.

Let A and B be arbitrary sets. Let $f : A \rightarrow 2^B$ be a function that assigns to each element $a \in A$ a subset $f(a)$ of B, and let $C \subseteq A$. We define

$$f[C] = \bigcup_{c \in C} f(c). \qquad (*)$$

Hence, $f[C] \in 2^B$. For a given set A, A^N denotes the set of all *vectors* $\vec{a} = (a_1, a_2, \ldots, a_N)$, where

$a_i \in A$ for every i.

Definition: Let X and D be sets of *input values* and *decision values*, respectively. A *distributed decision task* T is a function

$$T: X_T \rightarrow 2^{D^N} - \{\emptyset\},$$

where $X_T \subseteq X^N$. X_T is called the *input set* of the task T. The *decision set* of the task T is the set $D_T = T[X_T]$. Each vector $\vec{x} = (x_1, x_2, ..., x_N) \in X_T$ is called an *input vector*, and it represents the initial assignment of the *input value* $x_i \in X$ to processor P_i, for $i = 1, 2, ..., N$. Each vector $\vec{d} = (d_1, d_2, ..., d_N) \in D_T$ is called a *decision vector*, and it represents the assignment of a *decision value* $d_i \in D$ to processor P_i, for $i = 1, 2, ..., N$.

Thus, a decision task T maps each input vector to a non-empty set of allowable decision vectors. We assume that all tasks T discussed in this paper are *computable*, in the sense that the set $\{(\vec{x}, \vec{d}) : \vec{x} \in X_T, \vec{d} \in T(\vec{x})\}$ is recursive.

Examples:

(1) **Consensus** [FLP]: A consensus task is any task T where $X_T = X^N$ for an arbitrary set X, and such that $T(\vec{x}) \subseteq \{ (0,0,...0), (1,1,...,1) \}$ for every input vector $\vec{x} \in X_T$. In the sequel, $\vec{0}$ denotes the vector $(0,0,...,0)$, and $\vec{1}$ denotes the vector $(1,1,...,1)$. A *strong* consensus task is a consensus task T, in which there exist two input vectors $\vec{u}$ and $\vec{v}$ such that $T(\vec{u}) = \{\vec{0}\}$ and $T(\vec{v}) = \{\vec{1}\}$. The main result in [FLP] implies that the strong consensus task T, with $X_T = \{0,1\}^N$, is not 1-solvable. A *weak* consensus task is a consensus task that is not strong.

(2) **Approximate Consensus** [DLPSW]: This task is defined for any given $\varepsilon > 0$. The input set X_T is Q^N, where Q is the set of rational numbers, and for a given input $\vec{x} = (x_1, \cdots, x_N)$, $T(\vec{x})$ is the set of all vectors $\vec{d} = (d_1, \cdots, d_N)$ satisfying $|d_i - d_j| \leq \varepsilon$ and $m \leq d_i \leq M$ $(1 \leq i, j \leq N)$, where $m = \min \{ x_1, \cdots, x_N \}$ and $M = \max \{ x_1, \cdots, x_N \}$.

(3) **Order Preserving Renaming (OPR)** [ABDKPR]: This task is defined for a given integer K, where $K \geq N$. The input set X_T is the set of all vectors $(x_1, \cdots, x_N)$ of distinct integers. For a given input $\vec{x}$, $T(\vec{x})$ is the set of all integer vectors $(d_1, \cdots, d_N)$ satisfying $1 \leq d_i \leq K$ and such that for each i, j, $x_i < x_j$ implies $d_i < d_j$.
Note that the model in [ABDKPR] assumes that the processors do not have identities. As mentioned above, our results can be modified to hold for this model too.

2.3 Protocols and Executions

A *protocol* α for a given network is a set of N programs, each associated with a single processor in the network. Each such program contains operations of sending a message to a neighbor, receiving a message and processing information in the local memory.

If the network is initialized with the input vector $\vec{x} \in X^N$ (i.e., the value x_i is assigned to processor P_i), and if each processor executes its own program in α, then the sequence of operations performed by the processors is called an *execution of* α *on input* $\vec{x}$. For this definition, we assume that no two operations occur simultaneously; otherwise, we order them arbitrarily. For more formal definitions see, e.g.,

[FLP, KMZ]. Note that an execution on a given input is not necessarily unique, due to the asynchrony in the network. The set of all the executions of a protocol α on an input $\vec{x}$ is denoted by $E_\alpha(\vec{x})$.

Definition: A *complete execution e* of a protocol α on input $\vec{x}$ is an execution of α on input $\vec{x}$ in which all the processors eventually *decide*, by writing a decision value in a write-once register. The vector $\vec{d} = (d_1, d_2, ..., d_N)$, where d_i is the decision value of processor p_i, for every i, is called the *output vector of the execution e*, and is denoted by $D_\alpha(e, \vec{x})$. $D_\alpha(\vec{x})$ is the set of all output vectors of all the complete executions of the protocol α on input $\vec{x}$. Namely, $D_\alpha(\vec{x}) = \bigcup_{e \in CE_\alpha(\vec{x})} D_\alpha(e, \vec{x})$, where $CE_\alpha(\vec{x})$ is the set of all complete executions of α on $\vec{x}$. For a set S of input vectors, $D_\alpha[S]$ is defined according to (*).

Note that the definition above does not require the processors to halt after reaching a decision. However, in our universal protocol, deciding will always be associated with halting.

2.4 Solvability and 1-Solvability

Definition: A protocol α *solves* a task T if for every input vector $\vec{x} \in X_T$, it satisfies:

(1) $E_\alpha(\vec{x}) = CE_\alpha(\vec{x})$ (i.e., all the executions of α on $\vec{x}$ are complete), and

(2) $D_\alpha(\vec{x}) \subseteq T(\vec{x})$ (i.e., each execution of α on $\vec{x}$ results in a legal output vector).

Note that for every (computable) task T there is a protocol that solves it, by first having each processor send its input to a specified processor, say P_1, and then letting P_1 decide on some vector $\vec{d}$ in $T(\vec{x})$ and broadcast it to all other processors.

Definition: A processor P is *faulty* in an execution e if all the messages sent by P during e after a certain time are never received (a *crash* failure).

Next we define the notion of solvability in spite of one fault. We adapt the approach in [MW].

Definition: A protocol α *1-solves* a task T (and in this case, T is 1–*solvable*) if the following two conditions hold:

(1) if no processor is faulty then α solves T, and

(2) if, in an execution e, one processor is faulty, then all other processors eventually decide.

The strong consensus tasks are shown in [FLP] not to be 1-solvable. The weak consensus tasks are clearly 1-solvable, by simply letting every processor decide on 1 (or 0), regardless of the input. The reason is that our definition of solvability does not exclude such trivial solutions. Note that our definition differs from the ones in [FLP, MW] in that we do not require a protocol that solves T to achieve every possible output. We use this definition since we believe it is more natural: usually, one considers a task solved by any protocol that always outputs an acceptable decision vector, regardless of whether there are some acceptable decision vectors that are never achieved.

3. TWO BASIC CONDITIONS FOR 1-SOLVABILITY

Given a task T and a protocol α, we present in this section two necessary conditions for the 1-solvability of T by α.

Intuitively, the first of these conditions, called the *connectivity condition*, reflects the fact that a processor in the network may fail immediately after it "chooses" one out of some decision values, leaving the rest of the processors in doubt as to its actual decision value. The second condition, called the *extendibility condition*, reflects the fact that a processor may fail before starting the computation, forcing the other processors to reach a decision without knowing its input value. These conditions are later used to obtain a complete characterization of 1-solvable tasks.

3.1 The Connectivity Condition

Definition: Let $S \subseteq A^N$, for a given set A. Two vectors $\vec{s}_1, \vec{s}_2 \in S$ are *adjacent* if they differ in exactly one entry. The *adjacency graph of S*, $G(S) = (S, E_s)$, is an undirected graph, where $(\vec{s}_1, \vec{s}_2) \in E_s$ iff $\vec{s}_1$ and $\vec{s}_2$ are adjacent. For a task T, the adjacency graph $G(X_T)$ of the input set X_T is called the *input graph* of T. The *decision graph* $G(D_T)$ of T is defined similarly.

Our first theorem is a straightforward generalization of Theorem 3.5 in [MW], which we state here using our notation:

Theorem MW: Let T be a decision task that has a connected input graph, and let α be a given protocol. If α 1-solves T, then $G(D_\alpha[X_T])$ is connected. □

Our first theorem extends Theorem MW to cases where only a sub-task of the given task satisfies the assumption of that theorem.

Theorem 1 (The Connectivity Condition): Let T be a decision task, let $C \subseteq X_T$ be such that $G(C)$ is a connected subgraph of the input graph $G(X_T)$, and let α be a given protocol. If α 1-solves T, then $G(D_\alpha[C])$ is connected.

Proof. Let α be a protocol that 1-solves the task $T: X_T \to D_T$. Define a new task $T': C \to D_\alpha[C]$, such that $X_{T'} = C$ and $T'(\vec{x}) = D_\alpha(\vec{x})$ for every $\vec{x} \in C$. α clearly 1-solves T', and by applying Theorem MW to T' we have that $G(D_\alpha[X_{T'}])$ is connected. □

We shall use in the sequel the following corollary of Theorem 1, for which we need the following definitions:

Definition: Let T be a task and α a protocol which solves T. T^α is the task *induced* by α and X_T; that is: $X_{T^\alpha} = X_T$, and $T^\alpha(\vec{x}) = D_\alpha(\vec{x})$ for every $\vec{x} \in X_{T^\alpha}$.

Definition: A task T is *pointwise connected* if $G(T(\vec{x}))$ is connected for each $\vec{x} \in X_T$.

Corollary 1: If a protocol α 1-solves a task T then T^α, the task induced by α and X_T, is pointwise connected.

3.2 The Extendibility Condition

Our next theorem is based on the following observation: Consider an execution of a protocol that 1-solves a given task, in which we delay all the messages sent by any processor P for long enough.

Then, eventually, all other processors must decide. Moreover, the decisions they make (knowing only $N-1$ input values) must be extendible to an acceptable decision vector. (A similar observation was used in the proof of Lemma 6.1 in [ABDKPR].)

We need the following definitions for our discussion:

Definition: A *partial vector* is a vector in which one of the entries is not specified; this entry is denoted by '$*$'. For a vector $\vec{s}=(s_1, \cdots, s_N)$, $\vec{s}^i$ denotes the partial vector obtained by assigning $*$ to the i-th entry of $\vec{s}$, i.e., $\vec{s}^i = (s_1, \cdots, s_{i-1}, *, s_{i+1}, \cdots, s_N)$. $\vec{s}$ is called an *extension* of $\vec{s}^i$. For a set of vectors S, $S^i = \{\vec{s}^i : \vec{s} \in S\}$.

Definition: Let $\vec{x}^i$ be a partial input vector and $\vec{d}^i$ a partial decision vector of a task T. $\vec{d}^i$ is a *covering vector* for $\vec{x}^i$ if for each extension of $\vec{x}^i$ to an input vector $\vec{x} \in X_T$, there is an extension of $\vec{d}^i$ to a decision vector $\vec{d} \in T(\vec{x})$.

Definition: Let α be a protocol that 1-solves a task T. An *i-sleeping execution* of α is an execution in which *all* the messages sent by P_i are delayed until all other processors decide (such an execution exists by the definition of 1-solvability, since P_i is not distinguishable from a faulty processor).

Theorem 2 (The Extendibility Condition): Let T be a decision task and α be a protocol that 1-solves T. Then in T^α, the task induced by α and X_T, there is a covering vector for each partial input vector.

Proof: Let $\vec{x}^i = (x_1, \cdots, x_{i-1}, *, x_{i+1}, \cdots, x_N)$ be a partial input vector. Consider an i-sleeping execution of α, in which the input to P_j is x_j for each $j \neq i$. Let $\vec{d}^i = (d_1, \cdots, d_{i-1}, *, d_{i+1}, \cdots, d_N)$ be the partial vector output by the non-sleeping processors. We show that $\vec{d}^i$ is a covering vector for $\vec{x}^i$.

Let the vector $\vec{y} = (x_1, \cdots, x_{i-1}, y_i, x_{i+1}, \cdots, x_N) \in X_T$ be an extension of $\vec{x}^i$. We have to show an extension of $\vec{d}^i$ to a decision vector in $T^\alpha(\vec{y})$. For this, assume that $\vec{y}$ is the actual input to α, and that P_i is eventually awakened. P_i must eventually decide on a value d_i to obtain an output vector $\vec{d} = (d_1, \cdots, d_i, \cdots, d_N)$. This $\vec{d}$ is the desired extension of $\vec{d}^i$. □

The vector $\vec{d}$ in the proof above will play an important role in the sequel, and we give it a formal definition:

Definition: A vector $\vec{d}$ is an *i-anchor* of an input vector $\vec{x}$ if $\vec{d} \in T(\vec{x})$ and $\vec{d}^i$ is a covering vector for $\vec{x}^i$.

Let $T_E(\vec{x}^i)$ denote the set of all covering vectors for $\vec{x}^i$. It is not difficult to see that $T_E(\vec{x}^i) = \cap \{ (T(\vec{y}))^i \mid \vec{y} \in X_T \text{ is an extension of } \vec{x}^i \}$.

Examples: Let T be the OPR task for $N=3$ processors and $K=4$. Consider the partial input vector $\vec{v}^3 = (10,12,*)$ and the following input vectors which are extensions of $\vec{v}^3$: $\vec{x}_1 = (10,12,13)$, $\vec{x}_2 = (10,12,11)$, $\vec{x}_3 = (10,12,9)$. By the definition of T,

$T(\vec{x}_1) = \{ (1,2,3), (1,2,4), (1,3,4), (2,3,4) \}$,

$T(\vec{x}_2) = \{ (1,3,2), (1,4,2), (1,4,3), (2,4,3) \}$ and

$T(\vec{x}_3) = \{ (2,3,1), (2,4,1), (3,4,1), (3,4,2) \}$.

Hence

$(T(\vec{x}_1))^3$ = { (1,2,∗), (1,3,∗), (2,3,∗) },

$(T(\vec{x}_2))^3$ = { (1,3,∗), (1,4,∗), (2,4,∗) },

$(T(\vec{x}_3))^3$ = { (2,3,∗), (2,4,∗), (3,4,∗) }

and $(T(\vec{x}_1))^3 \cap (T(\vec{x}_2))^3 \cap (T(\vec{x}_3))^3 = \varnothing$. This implies that $T_E(\vec{v}^3) = \varnothing$, so by Theorem 2 this task is not 1-solvable.

Now examine the same task with N=3 and K=5. For the same three input vectors we have

$(T(\vec{x}_1))^3$ = { (1,2,∗), (1,3,∗), (2,3,∗), (1,4,∗), (2,4,∗), (3,4,∗) },

$(T(\vec{x}_2))^3$ = { (1,3,∗), (1,4,∗), (2,4,∗), (1,5,∗), (2,5,∗), (3,5,∗) } and

$(T(\vec{x}_3))^3$ = { (2,3,∗), (2,4,∗), (3,4,∗), (2,5,∗), (3,5,∗), (4,5,∗) }.

Then $(T(\vec{x}_1))^3 \cap (T(\vec{x}_2))^3 \cap (T(\vec{x}_3))^3$ = {(2,4,∗)}. Indeed, in this task, if messages sent by some processor are delayed, then the other two processors may eventually decide on 2 and 4; no matter what is the input of the delayed processor, it can always extend the partial decision vector to a legal decision vector by deciding on 1, 3 or 5. In fact, in the example above, T_E((10,12,∗)) = { (2,4,∗) }, that is: (2,4,∗) is the only covering vector for (10,12,∗). Its extensions - (2,4,1),(2,4,3) and (2,4,5) - are 3-anchors of the input vectors (10,12,9),(10,12,11) and (10,12,13), respectively. (Note that in this example the covering vectors and the anchors are unique. This is not always the case, as can be exemplified by the OPR task with N =3 and K=6.)

One can easily extend this example to show that the minimum K required for the OPR task for N processors to be 1-solvable is $2N$-1 (N-1 decisions and N possible extensions). This result is proved in [ABDKPR], where it is also shown that this condition suffices for the task to be 1-solvable (actually, the result presented there is optimal for any number of faults).

4. NECESSARY AND SUFFICIENT CONDITIONS FOR 1-SOLVABILITY

In this section we combine the necessary conditions given in Theorems 1 and 2 to give a a necessary and sufficient condition for a task to be 1-solvable. In proving the positive direction we present a universal protocol which 1-solves any such task. First we need one more definition.

Definition: A task T' is a *restriction* of a task T if $X_{T'} = X_T$, and $T'(\vec{x}) \subseteq T(\vec{x})$ for every $\vec{x} \in X_{T'}$.

Note that, for a restriction T' of T, if a protocol α (1-)solves T', then α also (1-)solves T. Also note that if protocol α (1-)solves T, then T^{α} , the task induced by X_T and α, is a restriction of T.

Theorem 3: A task T is 1-solvable if and only if there exists a restriction T' of T satisfying:

(3a) T' is pointwise connected, and

(3b) For each partial input vector $\vec{x}^i$ there is a covering vector $\vec{d}^i$ in T'; moreover, there is a (centralized) algorithm that on input $\vec{x}^i$ outputs such a $\vec{d}^i$.

Proof: First we use the results of the previous section to prove the “only if” part of the theorem. Then we prove the “if” part, by presenting a universal protocol that 1-solves any task satisfying (3a) and (3b).

Only if: Let α be a protocol that 1-solves T. Take T' to be T^{α}, the task induced by α and X_T. T^{α} is pointwise connected by Corollary 1, and by Theorem 2 it contains a covering vector for each partial input vector $\vec{x}^i$; moreover, for each partial input vector $\vec{x}^i$, the corresponding $\vec{d}^i$ can be computed by simulating an i-sleeping execution of α on input $\vec{x}^i$, as described in the proof of Theorem 2.

Before proving the **if** part, we give an example to illustrate the proof above: Consider the OPR task with N=3 and K=5, which as mentioned above is 1-solvable. We show that if we add to it the requirement that each decision vector must have 2 as one entry and 4 as another entry, then the resulting task is not 1-solvable.

Consider the input vector $\vec{x}$ = (10,20,30). By the definition of this task, $T(\vec{x}) = \{(1,2,4),(2,3,4),(2,4,5)\}$. Consider the 3 partial input vectors $\vec{x}^1 = (*,20,30)$, $\vec{x}^2 = (10,*,30)$ and $\vec{x}^3 = (10,20,*)$. To each of these partial input vectors there is a unique covering vector in T ($\vec{d}^i$ denotes the covering vector for $\vec{x}^i$): $\vec{d}^1 = (*,2,4)$, $\vec{d}^2 = (2,*,4)$ and $\vec{d}^3 = (2,4,*)$. Therefore, for any restriction T' of T that satisfies condition (3b), it must hold for $i = 1,2,3$ that there is some extension of $\vec{d}^i$ in $T'(\vec{x})$. This implies that $T'(\vec{x}) = T(\vec{x})$. But $G(T(\vec{x}))$ is not connected, and hence T' is not pointwise connected. It follows that this task is not 1-solvable.

We now complete the proof of Theorem 3:

If: Let T be a task which has a restriction T' satisfying (3a) and (3b). (An example of a solvable task is the OPR task for $N = 3$ and K =5, with $T' = T$; see Figure 1b.) We will present a protocol which 1-solves T', and hence 1-solves T.

By condition (3b), there is an algorithm COMP.COVER that gets as an input a partial input vector $\vec{x}^i$ and outputs a partial covering vector for $\vec{x}^i$, $\vec{d}^i$. By (3a) $G(T'(\vec{x}))$ is connected. Hence, for a given finite set $S_{\vec{x}}$ of i-anchors of $\vec{x}$, there is a *finite* tree $TR_{\vec{x}}$ in $G(T'(\vec{x}))$ that contains $S_{\vec{x}}$. It is not hard to show, by the computability of T, that there is an algorithm COMP.TREE that on input $\vec{x}$ and a (finite) set $S_{\vec{x}}$ of i-anchors of $\vec{x}$, outputs a tree $TR_{\vec{x}}$ as above and a root $r_{\vec{x}}$, which is an (arbitrary) vertex in $TR_{\vec{x}}$.

Our protocol assumes that each processor P_k has a copy of these algorithms COMP.COVER and COMP.TREE.

The general outline of the protocol is as follows: In the first two stages each processor P_k is trying to find out the input vector $\vec{x}$; for this, it first broadcasts its input value and receives $N-1$ input values (including its own), which determine a partial input vector $\vec{x}^i$ (note that $i \neq k$). Then it broadcasts this partial vector, and waits until it receives $N-1$ such partial vectors. If all these $N-1$ partial vectors are equal to its own, $\vec{x}^i$, then P_k *decides* on the partial output vector $\vec{d}^i$ = COMP.COVER($\vec{x}^i$) (by saying that P_k decides on a (partial) output vector $(d_1, \cdots, d_k, \cdots, d_N)$ we mean, in particular, that d_k is the decision value of P_k). Then P_k broadcasts its decision on $\vec{d}^i$ to all other processors, and halts. Otherwise, P_k knows the input vector $\vec{x}$, and it first computes N anchors $\vec{A}_1, \cdots, \vec{A}_N$, where $\vec{A}_i$ is an extension of $\vec{d}^i$ = COMP.COVER($\vec{x}^i$) to an i-anchor of $\vec{x}$ (it is assumed that all the processes compute the same extensions). then it computes the tree $TR_{\vec{x}}$ = COMP.TREE($\vec{x}, \vec{A}_1, \cdots, \vec{A}_N$). Note that the tree $TR_{\vec{x}}$ is a function of the input vector $\vec{x}$ alone, since each of the $\vec{A}_i$'s is determined by $\vec{x}$.

In the rest of the protocol each processor attempts to decide on a certain vertex in $TR_{\vec{x}}$, such that

eventually every processor will decide on one out of two adjacent vertices (vectors) (this guarantees that the actual output vector is one of these two vectors, and hence it is in $T(\vec{x})$). This part of the algorithm is done in phases, where in phase l a processor broadcasts a message in which it either *suggests* a certain vertex $\vec{d}$ as a possible decision, or informs the other processors that it has decided on some $\vec{d}$. If in phase $l+1$ it receives $N-1$ messages of phase l suggesting the same vertex $\vec{d}$, or at least one message deciding on $\vec{d}$, then it decides on $\vec{d}$, broadcasts its decision, and halts. Otherwise it picks up a vertex $\vec{d}'$ that was suggested in a maximal number of messages of stage l, and it suggests in phase $l+1$ the *father* of $\vec{d}'$ in $TR_{\vec{x}}$ as a possible decision. This process guarantees that if for long enough no vertex was decided upon, all the processors will eventually suggest the root $r_{\vec{x}}$ (which is defined to be its own father), and then will decide on it.

A formal description of the protocol is given below; in this protocol we use the procedures COMP.COVER and COMP.TREE described above. It is assumed in this protocol that $N > 3$. In the Appendix we show how to modify the protocol to work also when $N = 3$.

The protocol for P_k:

A. broadcast your input value x_k and wait until you receive $N-1$ stage-A messages.

B. Now for some j ($1 \le j \le N$) you know $\vec{x}^j$. (Note that $j \neq k$)
BROADCAST($\vec{x}^j$), and wait until you receive $N-1$ stage-B messages.
$l \leftarrow 1$ {l is the phase number}
if all the $N-1$ stage-B messages you received are equal to $\vec{x}^j$ **then**
begin
$\vec{d}^j \leftarrow$ COMP.COVER($\vec{x}^j$)
DECIDE($\vec{d}^j$) {your output value is the k-th entry of $\vec{d}^j$}
BROADCAST(DECIDE, $\vec{d}^j$);
HALT;
end;
else {now you know the input vector $\vec{x}$}
$TR_{\vec{x}} \leftarrow$ COMP.TREE($\vec{x}, \vec{A_1}, \cdots, \vec{A_N}$), where $\vec{A_i}$ is an extension of $\vec{d}^i$ = COMP.COVER($\vec{x}^i$) to an i-anchor of $\vec{x}$.
Let s be such that the partial vector $\vec{x}^s$ was received a maximal number of times in Stage B.
$\vec{d} \leftarrow father(\vec{A_s})$ {$\vec{d}$ gets the father, in $TR_{\vec{x}}$, of the s-anchor of $\vec{x}$}
BROADCAST(SUGGEST, $\vec{d}$, l);

C. *decided* $\leftarrow$ *false*
while NOT *decided* **do**
begin
$l \leftarrow l+1$;
RECEIVE $N-1$ messages of phase $l-1$; {(DECIDE, $\vec{d}$) messages are considered to be of phase m for every m}
if all $N-1$ messages are (SUGGEST,$\vec{d}'$, $l-1$) or one of the messages is a (DECIDE, $\vec{d}'$) {(DECIDE, $\vec{d}^s$) message of phase B is considered as a (DECIDE, $\vec{A_s}$) message, where $\vec{A_s}$ is the s-anchor that you computed in stage B} **then**
begin
DECIDE ($\vec{d}'$);
BROADCAST (DECIDE, $\vec{d}'$);
decided $\leftarrow$ *true*;
end
else let $\vec{d}'$ be a vertex that was suggested in a maximal number of phase $l-1$ messages.

$\vec{d} \leftarrow father(\vec{d}')$;
BROADCAST(SUGGEST, $\vec{d}$, l);

end.

The correctness of the protocol follows from the claims below. For a given input vector $\vec{x}$, $TR_{\vec{x}}$ is the tree computed by COMP.TREE; $L_{\vec{x}}$ denotes the maximal distance, in $TR_{\vec{x}}$, from an i-anchor to its root $r_{\vec{x}}$. Also, for a set of vertices U in $TR_{\vec{x}}$, $father(U)$ denotes the set $\{father(v) : v \in U\}$ (recall that $father(r_{\vec{x}}) = r_{\vec{x}}$). Clearly, $father^{L_{\vec{x}}}(U) = \{r_{\vec{x}}\}$ for any subset U of vertices of $TR_{\vec{x}}$ ($father^h$ is the function obtained by applying *father* repeatedly h times).

Claim 1: In each execution of the protocol on input $\vec{x}$, there is an $l \le L_{\vec{x}}+1$ such that at least one processor decides in phase l (a processor decides upon executing the function DECIDE, in lines 8 or -8 in the code above).

Proof: Let U_0 be the set of i-anchors of $\vec{x}$ in $TR_{\vec{x}}$, and for $l > 0$ let U_l be the set of vertices appearing in a SUGGEST message in phase l of this execution. Assume that no processor decides in any phase $l \le L_{\vec{x}}$. Then for each such l, each non-faulty processor will execute that last three lines of the code. This implies that each such processor will send a SUGGEST message in phase l, and that $U_{l+1} \subseteq father(U_l)$.

This implies, by the discussion above, that $U_{L_{\vec{x}}} = \{r_{\vec{x}}\}$, which means that in phase $L_{\vec{x}}$ all the non-faulty processors send the message (SUGGEST, $r_{\vec{x}}$,$L_{\vec{x}}$), and hence in phase $L_{\vec{x}}+1$ every processor will receive $N-1$ such messages, and hence will decide on $r_{\vec{x}}$. □

In claims 2-4 below, l_0 is the minimal l satisfying Claim 1, P_k is a processor that decides in phase l_0, and $\vec{d}$ is the vertex it decides on. In these claims, we consider a decision of P_k on a partial vector $\vec{d}^{\,i}$ at stage B as a decision of P_k on the anchor $\vec{A}_i$; such an assumption is possible since $i \neq k$, and hence the decision value of P_k is the same in both cases.

Claim 2: If some processor P_j decides in phase l_0 on a vertex $\vec{d}'$, then $\vec{d}' = \vec{d}$.

Proof: Since no processor decides in phase l_0-1, a processor decides in phase l_0 on a vertex $\vec{d}$ iff it receives $N-1$ messages of phase l_0-1 suggesting $\vec{d}$. The proof follows by the observation that, since $N > 2$, it is impossible to have two distinct vertices, each suggested by $N-1$ messages of phase l_0-1. □

Claim 3: One of the following holds:

(a) At least two processors send in phase l_0 a (DECIDE, $\vec{d}$) message, or

(b) All the (non-faulty) processors, except P_k, send in phase l_0 a (SUGGEST, $father(\vec{d})$,l_0) message.

Proof: Assume that (a) does not hold. Then exactly one processor, P_k, sends a (DECIDE, $\vec{d}$) message in phase l_0. Also, by the definition of l_0, P_k must have received $N-1$ (SUGGEST, $\vec{d}$, l_0-1) messages. Moreover, every other non-faulty processor P_j received exactly $N-2$ such messages and exactly one (SUGGEST, $\vec{d}'$, l_0-1) message, for some $\vec{d}'$.

Since $N-2 > 1$ (this is the only place where we use the fact that $N > 3$), P_j must send a (SUGGEST, $father(\vec{d})$, l_0) message, since $\vec{d}$ is the only vertex suggested by a maximal number of messages of phase l_0-1. □

Note: Claim 3 does not hold for $N = 3$. A modification of the algorithm that works also in this case is given in the Appendix.

Finally, from Claim 3 and step C of the protocol we get:

Claim 4: For $j = 1, \cdots, N$, every non-faulty processor P_j decides in phase l_0 or in phase l_0+1 on $\vec{d}$ or on *father*$(\vec{d})$.

Proof: If (a) of Claim 3 holds, then every non-faulty processor that has not decided on $\vec{d}$ in phase l_0 will receive at least one (DECIDE, $\vec{d}$) message in phase l_0+1, and hence will decide on $\vec{d}$ in this phase. Otherwise, case (b) of Claim 3 holds. This means that in phase l_0+1, every non-faulty processor will either receive $N-1$ (SUGGEST, *father*$(\vec{d})$, l_0+1) messages, and then will decide on *father*$(\vec{d})$, or will receive the (DECIDE, $\vec{d}$) of P_k, and will decide on $\vec{d}$. □

The proof of the correctness of the protocol is easily derived from the above claims and the observation that if all processors decide on two adjacent vertices then the vector they output is one of these vertices. This completes the proof of Theorem 3. □

We demonstrate the use of Theorem 3 by two examples, in which we extend two known 1-solvability results to their extremes. For this, all we have to do is to show existence of a restriction T' of T satisfying Theorem 3:

Strong Approximate Consensus: This task is defined similarly to the Approximate Consensus, with the following stronger restriction: For a given input $\vec{x} = (x_1, \cdots, x_N)$, let a be the average of the $N-1$ smallest x_i's, and let A be the average of the $N-1$ largest x_i's. $T(\vec{x})$ is restricted to the set of all vectors $\vec{d}=(d_1, \cdots, d_N)$ satisfying $|d_i - d_j| \le \varepsilon$ and $a \le d_i \le A$ $(1 \le i, j \le N)$. Note that this is the most severe restriction that can be imposed on the range of the output numbers. In fact, for a partial input vector $\vec{x}^{\,i} = (x_1, \cdots, x_{i-1}, *, x_{i+1}, \cdots, x_N)$ there is only one covering vector $\vec{d}^{\,i}$, which is the "constant" partial vector $(d, \cdots, *, \cdots, d)$, d being the average of the $N-1$ inputs $\{x_j : j \ne i\}$. It is not hard to verify that this task satisfies the conditions of Theorem 3 (with $T' = T$), and hence is 1-solvable.

We can further restrict this task, by requiring that the output vectors $\vec{d}$ are *monotone*, that is, $d_i \le d_{i+1}$. The resulting decision graph still satisfies Theorem 3, and hence the resulting task, called the *monotone strong approximate consensus*, is still 1-solvable. Moreover, by imposing the additional requirement that each decision vector $\vec{d}$ contains at most two distinct values (i.e., $\vec{d}$ = $(b, \cdots, b, c, \cdots, c)$, $b \le c \le a+\varepsilon$), we obtain a task which is still 1-solvable.

Restricted OPR with $N = 3$ **and** $K = 5$: This version is similar to the *OPR*, but it is required that the difference between the maximal and minimal entries of the output vector never exceeds 3. In fact, we can restrict the decision set for each input vector to include only 5 decision vectors. Figure 1a shows the corresponding graph $G(T(\vec{x}))$ for $\vec{x} = (A, B, C)$ with $A < B < C$. The three marked vectors are the anchors of such input. For comparison, Figure 1b shows the corresponding graph for the original formulation of the OPR problem. In fact, choosing any connected subgraph of this latter subgraph, that includes all the *i*-anchors, defines a version of the *OPR* which is 1-solvable. This example can be generalized to the *OPR* with N processors and $K = 2N-1$, to define a restricted version which allows only $2N-1$ possible output vectors for each input vector (see Figure 1c for the case N=4), compared to the $\binom{2N-1}{N}$ vectors which are allowed by the original formulation.

5. AN EXTENSION TO THE CASE OF UNKNOWN IDENTITIES

We sketch below the modifications needed in our definitions and proofs in order for our results to hold for the model in which the processors have no identities and the inputs are distinct (note that this includes the case where the processors have distinct identities which are not mutually known, and the input is arbitrary).

In this case N distinct input values are viewed as a subset of N elements of the input set X, and the input set of a task T, X_T, is a collection of such subsets of X.

Let D be the decision set. We denote by $C = C(X,D,N)$ the collections of all sets $\{(x_1,d_1), \cdots ,(x_N,d_N)\}$, where the x_i's are distinct elements of X and the d_i's are in D, $i = 1, \cdots ,N$.

A task T in this model is a function $T:X_T \rightarrow 2^C - \{\emptyset\}$. Informally, for each $\bar{x} = \{x_1,x_2,...,x_N\} \in X_T$, $T(\bar{x})$ is the collection of all sets $\{(x_1,d_1), (x_2,d_2), \ldots, (x_N,d_N)\}$ such that for input $\bar{x}$, the assignment of the decision value d_i to the processor whose input value is x_i ($i = 1, \cdots ,N$) is valid for T.

The decision set of T, $D_T = T[X_T]$ is defined as in section 2.2.

The vertices of the input and decision graphs, $G(X_T)$ and $G(D_T)$, are now sets of cardinality N, as described above. There is an edge connecting two such vertices iff they differ in exactly one element.

A *covering set for a partial input set* is defined similarly to the way a covering vector for a partial input vector was defined in Section 3.2, i.e., $\bar{d}^i$ is a *covering set* for a partial input set $\bar{x}^i$ if for each extension of $\bar{x}^i$ to an input set $\bar{x} \in X_T$, there is an extension of $\bar{d}^i$ to a decision set $\bar{d} \in T(\bar{x})$.

The equivalent of Theorem 3 in this case is:

Theorem 3': A task T is 1-solvable if and only if there exists a restriction T' of T satisfying the following:

(3'a) T' is pointwise connected, and

(3'b) For each partial input set $\bar{x}^*$ there is a covering set $\bar{d}^*$ in T'; moreover, there is a (centralized) algorithm that on input $\bar{x}^*$ outputs such a $\bar{d}^*$.

In the proof of Theorem 3', we first show that a task T satisfies conditions (3'a) and (3'b), iff by assigning distinct identities to the processors and using them to represent the task in the "vectorial" notation, we get a task that satisfies condition (3a) and (3b). This immediately implies the **only if** part.

In proving the **if** part, all that is needed is to adjust the operations in the universal protocol in Section 4 to the new definitions in a straightforward manner (i.e., change all occurrences of $\vec{x}$ to $\bar{x}$ etc.). It should be pointed out that, since the inputs are distinct, we may assume, as before, that all the processors that received all the input values will agree on the same anchors, and will construct the same tree $TR_{\bar{x}}$ used in the protocol.

6. LOWER BOUNDS

Once we have characterized the 1-solvable tasks, it is natural to consider the cost of their solutions. A natural measure for this cost is the *message complexity* of such a task, that is: the number of messages that must be sent in the worst case by any protocol that 1-solves it. Note that if all the processors are non faulty, then every computable task can be solved by $O(N)$ messages. In this section we use

the characterization theorem given in Section 4 to prove the following:

Theorem 4: For a given $N \geq 3$, there is a 1-solvable distributed task T for N processors satisfying the following: For every arbitrarily large constant M, there is an input $\vec{x}$ to T, such that every protocol that 1-solves T must send, in the worst case, at least M messages on input $\vec{x}$.

The proof of the above theorem is based on first showing that every protocol α that sends at most M messages on input $\vec{x}$ must satisfy that $|D_\alpha(\vec{x})| < F(M)$ for some function F, and then using Theorem 3 to show that if α 1-solves T then $|D_\alpha(\vec{x})| > F(M)$.

To simplify the discussion we consider in this section only executions that satisfy the FIFO discipline on each communication link. Clearly, if a protocol 1-solves a task T, then it must solve it also under this restricting assumption. The converse is also true, i.e., if a task T is 1-solvable by protocols that assume the FIFO discipline, then it is also 1-solvable by protocols that do not assume it, since this discipline can be simulated by having each processor P_i number the messages it sends to each processor p_j (note that for this simulation to work we need the assumption that once a message sent by P_i is lost, all the following messages sent by it are lost too). Hence, it is not hard to see that any lower bound that assumes this discipline is also applicable in the case where this discipline is not assumed.

Lemma 1: Let α be a protocol that 1-solves a task T, and let $\vec{x}$ be in X_T. If at most M messages are sent in any (FIFO) execution of α on $\vec{x}$, then $|D_\alpha(\vec{x})| < N^{2M}$. □

Proof: The sequence of M pairs (R,S), where the i-th pair denotes that the i-th message received in the execution (global time order) was received by P_R from P_S, uniquely defines a FIFO execution on input $\vec{x}$ with M messages. There are N^{2M} such different sequences, and this is an upper bound on the number of different FIFO executions on input $\vec{x}$ with M messages, which is an upper bound on $|D_\alpha(\vec{x})|$. □

Proof of Theorem 4: There exist tasks such that for each arbitrarily large M there exist an input vector $\vec{x}$, such that the distance between any 1-anchor and any 2-anchor of $\vec{x}$ is greater than N^{2M} (an example of such a task is given below).

Let T be such a task, let M be given, and let $\vec{x}$ be an input vector satisfying the above. Then by the proof of the **only if** part of Theorem 3 we know that every protocol α that 1-solves T must satisfy that $G(D_\alpha(\vec{x}))$ is connected and it contains an i-anchor of $\vec{x}$ for $i = 1,2,\cdots,N$. This implies that $|D_\alpha(\vec{x})| > N^{2M}$. By Lemma 1, this implies that α may send more than M messages on input $\vec{x}$. □

Example: One task for which the above lower bound is applicable is the Strong Approximate Consensus, for any fixed ε. To see this, let $\varepsilon = 1$, and consider an input $\vec{x} = (B(N-1), -B(N-1), 0, \cdots, 0)$, where B is some sufficiently large constant. Then every 1-anchor of $\vec{x}$ is of the form $\vec{A_1} = (A, -B, -B, \cdots, -B)$, where $|A + B| < \varepsilon$, and every 2-anchor of $\vec{x}$ is of the form $\vec{A_2} = (B, A', B, \cdots, B)$, where $|A' - B| < \varepsilon$, and the distance in $G(D_T)$ between any two such anchors is $2NB$. Since B can be taken to be arbitrarily large, this task has the desired properties.

Acknowledgement: We would like to thank the referees for their helpful comments.

REFERENCES

[ABDKPR] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, R. Reischuk "Achievable cases in an asynchronous environment", **Proceedings of the 28th FOCS**, October 1987, pp. 337-346.

[ASW] H. Attiya, M. Snir and M. Warmuth, "Computing on anonymous rings", **Journal of the ACM** 35 (4), 1988, pp. 845-875.

[BMZ1] O. Biran, S. Moran and S. Zaks, "Deciding 1-solvability of distributed tasks is NP-hard", in preparation.

[BMZ2] O. Biran, S. Moran and S. Zaks, "On the communication complexity of 1-solvable tasks", in preparation.

[DDS] D. Dolev, C. Dwork and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus", **Journal of the ACM** 34 (1), 1987, pp. 77-97.

[DLPSW] D. Dolev, N. A. Lynch, S. Pinter, E. Stark and W. Weihl, "Reaching approximate agreement in the presence of faults", **Journal of the ACM** 33 (3), 1986, pp. 499-516.

[Fe] A. D. Fekete, "Asynchronous Approximate Agreement" **Proceedings of the 6th ACM Symposium on Principles of Distributed Computing**, Vancouver, Canada, August 1987, pp. 64-76.

[FLM] M. J. Fischer, N. A. Lynch and M. Merritt, "Easy impossibility proofs for distributed consensus problems", **Distributed Computing, 1 (1), 1986, pp. 26-39.**

[FLP] M. J. Fischer, N. A. Lynch and M. S. Paterson, "Impossibility of distributed consensus with one faulty process", **Journal of the ACM** 32 (2), 1985, pp. 373-382.

[KMZ] E. Korach, S. Moran and S. Zaks, "Tight lower and upper bounds for some distributed algorithms for a complete network of processors", **Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing**, Vancouver, Canada, August 1984, pp. 199-207.

[LSP] L. Lamport, R. Shostak and M. Pease, "The Byzantine generals problem", **ACM Transactions on Programming Languages and Systems** 4 (3), 1982, pp. 382-401.

[MW] S. Moran and Y. Wolfstahl, "Extended impossibility results for asynchronous complete networks", **Information Processing Letters** 26, 1987/88, pp. 145-151.

[TKM] G. Taubenfeld, S. Katz and S. Moran, "Impossibility results in the presence of multiple faulty processors", submitted for publication, 1988.

APPENDIX: The Protocol for N ≥ 3

In the proof of Claim 3 we argued that if P_k receives $N-1$ messages suggesting a vertex $\vec{d}$, then every processor P_j that does not receive $N-1$ such messages (and hence does not decide on $\vec{d}$), must suggest $father(\vec{d})$ in the next phase. This is because P_j must have received $N-2$ messages suggesting $\vec{d}$, which is (since $N > 3$), more than the number of messages suggesting any other vertex.

The above argument does not hold when $N = 3$. Indeed, in this case there are scenarios where our protocol fails. In order for Claim 3 to hold also when $N = 3$, the protocol has to be modified in the two places where a processors may reach a decision, in stages B and C. We describe below the modification needed in stage C (the other modification is similar).

The idea is to replace each phase in the original protocol by a *double-phase*, which consists of two sub-phases: the first sub-phase is similar to a phase in the original protocol, with the following exception: upon receiving $N-1$ copies of some message (SUGGEST, $\vec{d}$, l), P_k does not decide on $\vec{d}$, but sends a message of the second sub-phase suggesting $\vec{d}$. Otherwise P_k sends a Null message at the second sub-phase. This guarantees that at most one vertex will be suggested at the second sub-phase of each phase.

If P_k receives $N-1$ messages of the second sub-phase suggesting $\vec{d}$, then it decides on $\vec{d}$. Otherwise, if it receives any message of the second sub-phase suggesting $\vec{d}$, then it suggests $father(\vec{d})$ at the first sub-phase of the following phase. If all the $N-1$ messages it receives in the second sub-phase are Null, it suggests at the first sub-phase of the next phase the father of a vertex that was suggested in a maximal number of messages of the first sub-phase.

It is easy to see that if P_k receives $N-1$ messages suggesting a vertex $\vec{d}$ at the second sub-phase, then every other processor will receive a message of the second sub-phase suggesting $\vec{d}$, and no other vertex will be suggested at this sub-phase. This guarantees that at the next phase every processor that has not yet decided on $\vec{d}$, will suggest $father(\vec{d})$, as claimed.

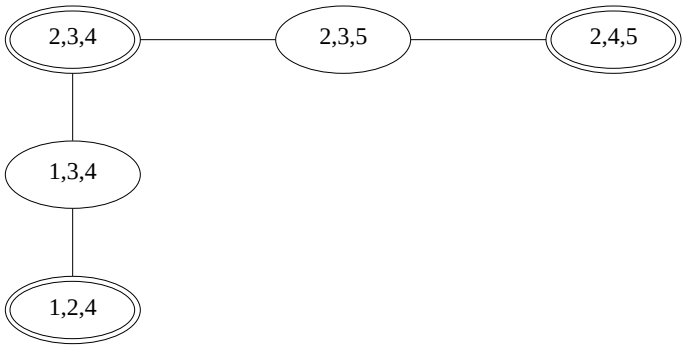

Figure 1a: $G(T(\vec{x}))$ in the restricted OPR task (N=3, K=5) for $\vec{x}=(A,B,C)$ s.t. $A < B < C$. The marked vertices are the i-anchors.

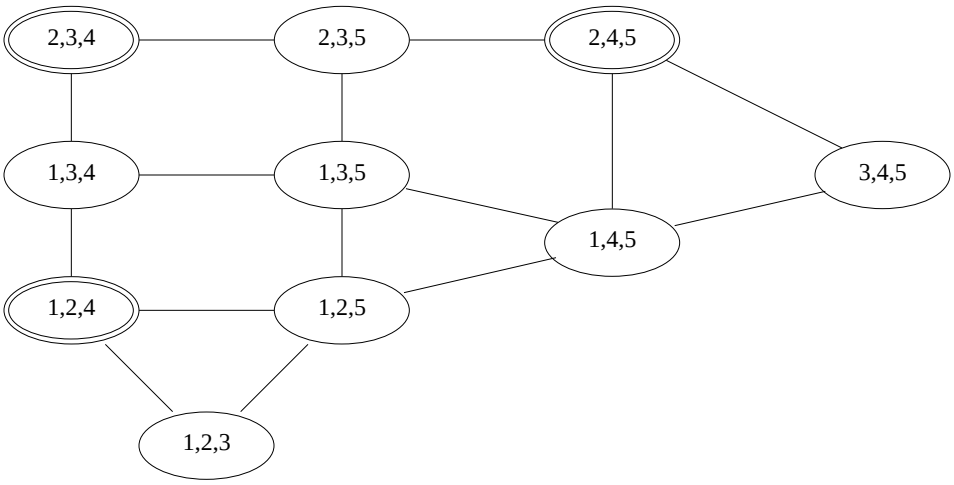

Figure 1b: $G(T(\vec{x}))$ in the original OPR task (N=3, K=5) for $\vec{x}=(A,B,C)$ s.t. $A < B < C$. The marked vertices are the i-anchors. Any connected subgraph that includes the three i-anchors defines a 1-solvable restricted version of the OPR task.

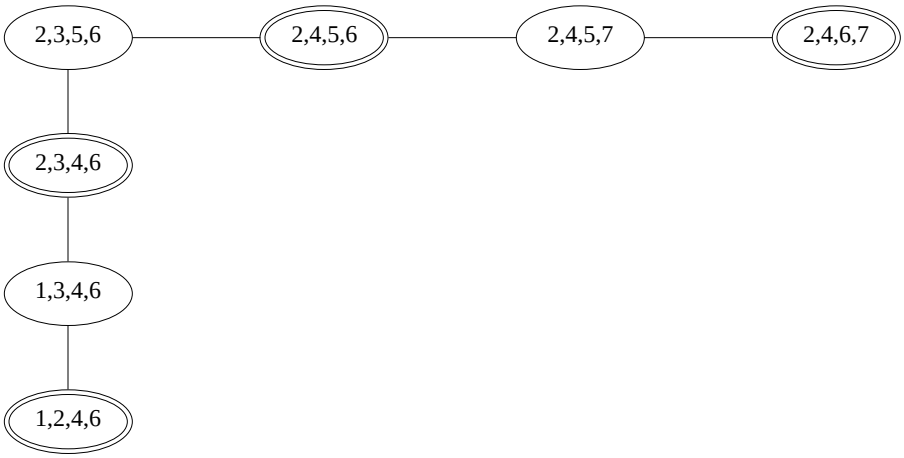

Figure 1c: The similar restricted OPR task for N=4 (K=7).