RESUMEN
A dependency graph is a graph that represents dependencies between objects of some application domain. They are directed graphs. Relations between nodes usually represent some form of dependency between objects. Let the nodes represent packages of a software system. An arrow from package A to package B may mean that package B depends on package A. Using dependency graph algorithms it is possible to compute properties of the nodes in the graph. For instance, for the given example, we could compute the sets of packages that are mutually dependent, or if they all depend on only one package. We could also compute if there are circular dependencies between packages. This is usually a problem when developing such mutually dependent packages. The paper will present the concepts related to dependency graphs and some algorithms with application to software engineering contexts, such as the one described in this abstract. The paper will also discuss some implementations details in a concrete programming language.

Palabras Clave
dependency graph, adjacency matrix, warshall algorithm, transitive closure, graph applications, software engineering

1. INTRODUCTION
A dependency graph is a graph that represents dependencies between objects of some application domain. It is a directed graph. Relations between nodes usually represent some form of dependency between objects. There are many applications of this concept in software engineering contexts. Some of those contexts relate to automating the construction of a product and some relate to the quality of the source code being produced.

As a quick example, consider packages in a software system. We have one package A that uses (or imports) another package B. If we represent a package as a vertex in the graph, and as an arrow from A to B the fact that package A is using package B, we are already modeling the problem using a dependency graph. The use of dependency graph algorithms will make possible to compute some properties of the vertices in the graph. For instance, for the given example, we could compute the sets of packages that are mutually dependent. We can get into some problems when we find such mutually dependent packages, and so we prefer to detect this situation to avoid it if possible. If it is not possible to avoid it, at least we are alerted of such dependency, and which elements are involved in this situation.

The paper presents some application contexts of such dependency graphs in sections 2 and 3. In section 4 the formal concepts related to dependency graphs are presented. Then a classical algorithm to compute the dependencies between all vertices in the graph and some implementation details are shown in section 5. The last section contains some conclusions.

2. A PROBLEM WITHIN THE BUILD PROCESS
In this section and the next we present two common software engineering contexts where we find tools that use dependency information. This section is related to tools that automate the build process. All this family of tools need dependency information to manipulate sequences of actions and objects created to manage the software package being constructed. It is a very typical software engineering problem that needs to be addressed through automated tools when large software systems are built.

For the example addressed in this section we have chosen almost randomly a tool. We could have chosen any, a more modern one (like graddle) or an old one (like make). They all base their work on scripts that contain dependency information. We ask the reader to concentrate on the modeling principles rather than the technology.

The next section contains some discussion on the applications of dependency graphs to static analysis tools.

But first, lets review the concept of dependency graph. We’ll need it when modeling dependency information in the application context.

A dependency graph is a directed graph used to represent dependencies between the vertices. A directed graph consists of a set of vertices and a set of arcs. Every arc relates two
2.2 A dependency graph for our sample ant script

We can build a dependency graph for our sample ant script. We proceed as follows:

1. let us associate with every target a vertex in the dependency graph,
2. besides, let us draw an arc from the source of the dependency to the target (reverse reading the ant script information).

If we label the vertices: I for target init, C for compile, D for dist and L for clean we can depict the dependency graph shown in figure 1.

By applying a well known algorithm on the dependency graph we could compute all the dependencies of the vertices. The dependencies are either direct or they are generated from the transitivity of the relation. Using this information we know that to execute the target dist we need to execute first init, then compile and finally dist.

3. EXAMPLES FROM STATIC ANALYSIS

Another common application context for dependency graphs is static analysis. Any compiler already does static analysis before generating code. Sometimes the compiler needs to check whether some definitions depend on each other. The dependency is not always direct, i.e. there is no arc from a vertex to itself. The dependency can be transitive and thus there can be many arcs between a vertex and itself.

Let’s go for some concrete examples in compiler technology. Consider Java packages or C# namespaces. Using one package from another is a form of dependency between packages. Consider methods. When a method calls another, the caller depends on the method being called. Consider types in a functional language like Haskell. Types can be recursive, this is they depend on each other. Their kinds should be computed to verify the consistency of the type definition. These are some examples of dependencies a compiler needs to discover from the source code either to give a precise diagnostic of an error or to generate code.

Besides compilers there are many other tools that can do static analysis of source code to compute properties of a program. Consider for instance a tool like NDepend\(^4\) for

\(^{1}\)The example has been taken from the ant documentation site: http://ant.apache.org/manual/using.html.

\(^{2}\)A target in ant is a named sequence of tasks to be executed.

\(^{3}\)Ant in progress build branch version 1.9.3.

\(^{4}\)Refer to the documentation of NDepend in
Visual Studio. It has features such as:

1. It shows the dependency graph between .net assemblies, allows to navigate through the graph browsing the types of the functions being defined. In such a dependency graph, vertices are assemblies and arcs denote the dependency between two assemblies.

2. It shows the call graph. A call graph shows the dependencies between instructions: the execution of an instruction is followed by the execution of the next instruction in the sequence, unless there is a conditional that will allow to follow one out two possible paths, or a loop that will repeat the execution of a sequence of sentences. In such a call graph, vertices are sentence identifiers and arcs represent possible execution paths.

Out of the information returned by these tools it is possible to

1. Track the execution of a program using the call graph.
2. Understand some design properties of an object-oriented program. For instance the dependencies between classes or packages give information about coupling. The dependencies between members of a class or packages give information about the cohesion of a class or a package.

4. TRANSITIVE CLOSURE OF A GRAPH

4.1 Dependency is a transitive relation

We will require that the dependency relation we are discussing for our applications is transitive. This means that if vertex $B$ depends on $A$ and if vertex $C$ depends on $B$, the conclusion that vertex $C$ depends on $A$ should follow naturally. We have seen this in the example of the ant script. To execute `dist` we need to execute `compile` and to execute `compile` we need to execute `init`, thus to execute `dist` we need to execute `init`.

The transitivity of the relation depends on the semantics of the arcs in the problem domain. We need to make sure that, if $A$, $B$ and $C$ are vertices in the set of vertices of the graph, if there is an arc between $A$ and $B$ and an arc between $B$ and $C$, it should make sense to read as if there were an arc from $A$ to $C$. Just as the case we revised in the previous paragraph for the ant script.

Let us define `path`. A path between two vertices in the graph is a sequence of arcs built by the application of the transitive relation. Consider the example in figure 2. We can see there is an arc between vertices $A$ and $B$, and thus there is a path between $A$ and $B$. The length of the path is 1. Length is defined as the number of arcs between two vertices. In the figure there is also a path between $C$ and $G$. The length of a path from $C$ to $G$ is 5. It is the minimum path between both vertices. Actually there are infinite paths between $C$ and $G$ because there is a cycle between $B$, $C$ and $D$, and another cycle between $E$ and $F$. A cycle shows a circular dependency relation between vertices. This usually means that all vertices in a cycle have a path that starts in a vertex and ends in the same vertex.

As is shown in figure 2 we can have vertices that do not depend on any vertex, like $A$. We also have vertices like $G$ that only depend on other vertices but no one depends on them. Finally we have “lonely” vertices such as $H$. These are vertices that are not related to any other vertex in the graph.

4.2 Graph representations

Although pictorial representations of graphs can be used to analyze some problems, for large graphs and/or for computer processing more formal representations are needed. We will use two representations in this paper: adjacency matrices and adjacency lists. In this section we will discuss the former. In the coming sections we will discuss the latter, specially when discussing alternative data structures that can be used to represent graphs.

An adjacency matrix $A$ is a boolean matrix\(^5\) used to represent a graph. Rows and columns can be enumerated using vertex names. The matrix element $A_{ij}$ will contain a 1 (true value) if there is an arc from vertex $V_i$ to vertex $V_j$, otherwise it contains a 0 (false value).

Figure 3 shows the adjacency matrix corresponding to the example depicted in figure 2. Notice that the intersection of row $C$ with column $D$ contains a value of 1. This is because there is an arc from $C$ to $D$.

Next, let us define sum and product of boolean matrices as follows:

- The sum of two boolean matrices $A$ and $B$ is the logical

\(^5\)A boolean matrix is a matrix that can only contain boolean values as 0 and 1 or true and false.
\( \lor \) operation applied element to element:
\[
A + B = \{ c \mid c = A_{ij} \lor B_{ij}, \forall i, j \}
\]

- The product of two boolean matrices \( A \) and \( B \) is the boolean matrix defined as:
\[
A \ast B = \{ c \mid c = \bigvee_k A_{ik} \land B_{kj}, \forall i, j \}
\]

### 4.3 The transitive closure

The interpretation of the product operation has some important consequences for the goal of this paper. Let us do the following: multiply the sample graph dependency adjacency matrix with itself. We get the adjacency matrix:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The resulting graph has an arc from \( A \) to \( C \) which was not in the original graph. It resulted from discovering that there is a path of length two from \( A \) to \( C \). This path is the result of the product applying the matrix onto itself. We can see this by expanding the computation of this element:
\[
e_{02} = (e_{00} \land e_{01}) \lor (e_{01} \land e_{12}) \lor (e_{02} \land e_{22}) \lor (e_{03} \land e_{32}) \lor (e_{04} \land e_{42}) \lor (e_{05} \land e_{52}) \lor (e_{06} \land e_{62}) \lor (e_{07} \land e_{72})
\]

The equation is exploring all ways in which it is possible to build a path between \( A \) and \( C \) through an intermediate vertex in the graph. The only viable solution is to go through \( B \): \( e_{01} \land e_{12} \). In general the product results in all paths of length two between any two vertices.

Thus, if \( M \) is the adjacency matrix, \( M \ast M \) (this is \( M^2 \)) gives all paths of lengths two. The product of \( M^2 \) by \( M \) (this is \( M^3 \)) will give all paths of length three, we explore all paths of length three by composing paths of length two with paths of length one. Applying this process several times we have that \( M^n \) gives all paths of length \( n \). Furthermore, by adding all computed adjacency matrices we get a matrix that contains all paths of length \( \leq n \):
\[
M_T = \sum_{i=1}^{n} M^i
\]

This graph/adjacency matrix is called the transitive closure of a graph/adjacency matrix. It contains as many arcs as paths between two nodes are possible in all possible combinations originated from the initial arcs.

The transitive closure of our example graph is shown in figure 4. In this graph you can already see both cycles: \( B, C \) and \( D, E, F \) submatrices show connection between all nodes. You can also see that from vertex \( A \) you can reach almost all vertices except the one that is not connected, i.e. \( H \).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 4:** Transitive closure for graph in figure 2

A row corresponding to a vertex in the matrix gives the information about all vertices that depend on it. A column that corresponds to a vertex shows all the vertices that are dependent on it. If the vertices were targets of an ant file, we could see that to execute target \( F \) we need to execute \( A \) through \( E \). We’ll need to start from targets that do not have dependencies, this is targets whose column have all 0s, in the example only \( A \) and \( H \). But \( H \) is neither reachable nor you can move out from it (all row \( H \) has 0s).

Transitive closures should be familiar to most programmers. The concept is found also in formal language theory, specially in regular languages (or read “regular expressions”). We have the star operator whose definition follows closely the pattern of a transitive closure, except that sum and product are defined differently (the alternative operator and string concatenation respectively).

### 5. An Algorithm to Compute the Transitive Closure

The mathematical treatment of the problem shown in the previous section appeared many years ago in [2]. From the adjacency matrix representation of the graph and the iterative nature of the solution it is possible to devise the following algorithm (so called the Warshall algorithm) written in pseudo-Java:

```java
boolean[][] m; // graph
boolean[][] mT; // transitive-closure to be computed
int n; // number of nodes in the graph

// Iteration 0 is the original graph
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        mT[i][j] = m[i][j];
    }
}

// Apply the transitive closure operations
for(int k = 0; k < n; k++) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            mT[i][j] = mT[i][j] || (mT[i][k] && mT[k][j]);
        }
    }
}
```

The concept is found also in formal language theory, specifically in regular languages (or read “regular expressions”).
This and a couple of additional techniques are discussed. The element in the array that represents this vertex should contain a list with two elements:

The resulting graph of iteration $k$ contains all arcs of length $\leq k$ between vertices in the graph.

Line 13 does $n$ iterations to find all arcs of length $\leq n$. The algorithm as presented is $O(n^3)$; we have three nested loops of length $n$.

A slight optimization can be done by analysis of the inner boolean assignment. The left operand in the $k$th expression does not depend on the loop variable, it thus can be taken out. The subexpression depends on the existence of this arc: there is no way we can find a transitive relation if this arc does not exist, and so we can leave out the exploration of possible new paths that depend on it. The new main transitivity closure algorithm code block is as follows:

```java
// Apply the transitive closure operations
for(int k = 0; k < n; k++) {
    for(int i = 0; i < n; i++) {
        if(mT[i][k])
            for(int j = 0; j < n; j++) {
                if(mT[k][j]) mT[i][j] = true;
            }
    }
}
```

This optimization does not improve theoretically on the order of the algorithm as it still behaves as $O(n^3)$ for the worst case, but for practical cases, where the original graph has not too many connections\(^6\) the speedup is noticeable.

### 5.1 Graph data structures to improve the computation of the transitive closure

For real application purposes, although the algorithm is still time consuming, it is still usable since there aren’t many connections between the vertices. This consideration can save us space in the store of the graph and consequently in processing the elements of the graph.

In section 4.2 it was mentioned that there is another data structure used to represent dependency graphs. This structure is called the adjacency list representation. The graph is an array of lists, one array element for every vertex in the graph. Every element in the array will contain a list of vertices.

Let’s go back to our example in figure 2. Take vertex $B$. The element in the array that represents this vertex should contain a list with two elements: $C$ and $E$. A complete representation of this dependency graph is shown in figure 5.

This and a couple of additional techniques are discussed in [1] and can be studied by the interested reader.

---

\(^6\) The adjacency matrix is usually sparse as not all the vertices have arcs to all other vertices.

---

**Figure 5: Adjacency list representation for graph in figure 2**

so contains an empirical study of the algorithms, where the author shows the improvements that can be achieved.

### 6. CONCLUSIONS

The paper presents the concept and some results of dependency graph applications in a software engineering context. The concept is not new, neither the applications. Somehow the last 15 years we have witnessed a growing amount of tools that use dependency information in a software production environment. This is partly due to the increase of the computational power of desktop computers. The overall complexity growth of current computer applications makes it sometimes necessary to work with these tools. Computational systems nowadays can have thousands of classes and hundreds of thousands of lines of code that are produced by large teams. Tools are usually complex since they combine many features. Nevertheless the underlying concepts and algorithms are not that complex as it was possible to see in the paper. For many simple applications it is even possible to write your own tool.

A dependency graph is a tool on its own. It is a tool for thinking about some type of problems (or sometimes an aspect of a problem we are trying to solve). The formality of such a tool makes it easy to produce automated tools. It was a secondary goal of the paper to show the importance of relying on such tools in the software engineer’s daily work. A paper with a title similar to this one has been written long ago [3]. It describes tools to manipulate and synthesize programs. One of the key elements to create such tools were dependency graphs, and of course additional formal language-based tools. This shows that having underlying sound and solid formal theories makes it possible to build such sophisticated tools. It is expected and hoped that the widespread dissemination of such formalisms conform a solid foundation of future software engineering infrastructure and tools.

### 7. REFERENCIAS


---

\[ A \rightarrow \{ B \} \]
\[ B \rightarrow \{ C, E \} \]
\[ C \rightarrow \{ D \} \]
\[ D \rightarrow \{ B \} \]
\[ E \rightarrow \{ F \} \]
\[ F \rightarrow \{ E, G \} \]
\[ G \rightarrow \{ \} \]
\[ H \rightarrow \{ \} \]