Analyze Maven POMs

To address your query, I'll provide a solution for generating a dependency graph in DOT format by recursively finding all `pom.xml` files in a directory and analyzing their dependencies, without assuming a fixed Maven multi-module project structure like the one in the example (`project/module1/pom.xml`, etc.). The script will focus on collecting all `pom.xml` files first, then analyzing them to determine module dependencies and output the graph.

Solution Overview

Here's what the script will do: 1. **Recursively Collect `pom.xml` Files**: Traverse the specified root directory to find all `pom.xml` files, regardless of their location or structure. 2. **Parse Module Information**: Extract `groupId`, `artifactId`, and dependencies from each `pom.xml`. 3. **Handle `groupId` Inheritance**: If a `pom.xml` doesn't specify a `groupId`, inherit it from a `<parent>` declaration, if present. 4. **Identify Intra-Project Dependencies**: Match dependencies to modules within the project based on `groupId` and `artifactId`. 5. **Generate DOT Output**: Create a directed graph of dependencies between modules.

This approach doesn't rely on a predefined directory hierarchy or assume that parent `pom.xml` files are in parent directories.

Python Script

Below is the Python script tailored to your requirements:

```
import os
import sys
import xml.etree.ElementTree as ET

# Define the Maven POM namespace
NS = "{http://maven.apache.org/POM/4.0.0}"

# Cache for groupId to avoid redundant parsing
group_id_cache = {}

def get_group_id(pom_path, pom_map):
    """
    Extract the groupId from a pom.xml file, considering inheritance from parent.
```

```
    Args:
        pom_path (str): Path to the pom.xml file.
        pom_map (dict): Mapping of pom.xml paths to their parsed data.

    Returns:
        str: The groupId of the module.
    """
    if pom_path in group_id_cache:
        return group_id_cache[pom_path]

    tree = ET.parse(pom_path)
    root = tree.getroot()
    group_id_elem = root.find(NS + 'groupId')

    if group_id_elem is not None:
        group_id = group_id_elem.text.strip()
    else:
        # Check for parent declaration
        parent = root.find(NS + 'parent')
        if parent is not None:
            parent_group_id = parent.find(NS + 'groupId').text.strip()
            parent_artifact_id = parent.find(NS + 'artifactId').text.strip()
            parent_relative_path = parent.find(NS + 'relativePath')
            if parent_relative_path is not None and parent_relative_path.text:
                parent_pom_path = os.path.normpath(
                    os.path.join(os.path.dirname(pom_path), parent_relative_path.text)
                )
            else:
                # Default to parent directory if relativePath is omitted
                parent_pom_path = os.path.join(os.path.dirname(pom_path), '..', 'pom.xml')
                parent_pom_path = os.path.normpath(parent_pom_path)

            if parent_pom_path in pom_map:
                group_id = get_group_id(parent_pom_path, pom_map)
            else:
                raise ValueError(f"Parent POM not found for {pom_path}: {parent_pom_path}")
        else:
            raise ValueError(f"No groupId or parent specified in {pom_path}")
```

```
    group_id_cache[pom_path] = group_id
    return group_id

def get_artifact_id(pom_path):
    """
    Extract the artifactId from a pom.xml file.

    Args:
        pom_path (str): Path to the pom.xml file.

    Returns:
        str: The artifactId of the module.
    """
    tree = ET.parse(pom_path)
    root = tree.getroot()
    artifact_id_elem = root.find(NS + 'artifactId')

    if artifact_id_elem is None:
        raise ValueError(f"pom.xml must specify artifactId: {pom_path}")

    return artifact_id_elem.text.strip()

def get_dependencies(pom_path):
    """
    Extract the list of dependencies from a pom.xml file.

    Args:
        pom_path (str): Path to the pom.xml file.

    Returns:
        list: List of tuples (groupId, artifactId) for each dependency.
    """
    tree = ET.parse(pom_path)
    root = tree.getroot()
    dependencies = []

    for dep in root.findall(NS + 'dependencies/' + NS + 'dependency'):
        dep_group_id_elem = dep.find(NS + 'groupId')
        dep_artifact_id_elem = dep.find(NS + 'artifactId')
        if dep_group_id_elem is not None and dep_artifact_id_elem is not None:
```

```
            dep_group_id = dep_group_id_elem.text.strip()
            dep_artifact_id = dep_artifact_id_elem.text.strip()
            dependencies.append((dep_group_id, dep_artifact_id))

    return dependencies

if __name__ == '__main__':
    # Check command-line arguments
    if len(sys.argv) != 2:
        print("Usage: python script.py <root_directory>")
        sys.exit(1)

    root_dir = sys.argv[1]
    if not os.path.isdir(root_dir):
        print(f"Error: {root_dir} is not a directory")
        sys.exit(1)

    # Step 1: Recursively find all pom.xml files
    pom_files = [
        os.path.join(root, file)
        for root, _, files in os.walk(root_dir)
        for file in files if file == 'pom.xml'
    ]

    if not pom_files:
        print(f"No pom.xml files found in {root_dir}")
        sys.exit(1)

    # Step 2: Build a dictionary of all POMs for parent lookups
    pom_map = {pom_file: None for pom_file in pom_files}

    # Step 3: Extract module information
    modules = {}  # (groupId, artifactId) -> pom_path
    for pom_file in pom_files:
        try:
            group_id = get_group_id(pom_file, pom_map)
            artifact_id = get_artifact_id(pom_file)
            modules[(group_id, artifact_id)] = pom_file
        except ValueError as e:
            print(f"Warning: Skipping {pom_file} due to error: {e}")
```

```
            continue

    # Step 4: Analyze dependencies
    dependencies = set()
    for pom_file in pom_files:
        try:
            importer_group_id = get_group_id(pom_file, pom_map)
            importer_artifact_id = get_artifact_id(pom_file)
            importer_key = (importer_group_id, importer_artifact_id)
            deps = get_dependencies(pom_file)
            for dep_group_id, dep_artifact_id in deps:
                dep_key = (dep_group_id, dep_artifact_id)
                if dep_key in modules and dep_key != importer_key:
                    # Add dependency as (importer, imported) tuple using artifactId for simplicity
                    dependencies.add((importer_artifact_id, dep_artifact_id))
        except ValueError as e:
            print(f"Warning: Error processing dependencies in {pom_file}: {e}")
            continue

    # Step 5: Output in DOT format
    print('digraph G {')
    for from_module, to_module in sorted(dependencies):
        print(f'  "{from_module}" -> "{to_module}";')
    print('}')
```

How It Works

1. Command-Line Input

- Takes a single argument: `<root_directory>`, the starting point for the recursive search.
- Validates that it's a directory.

2. Finding `pom.xml` Files

- Uses `os.walk` to recursively traverse the directory tree and collect all `pom.xml` files into a list.

3. Parsing Module Information

- **Group ID (`groupId`)**:

  - Extracted from each `pom.xml`.
  - If not present, looks for a `<parent>` section and resolves the `groupId` from the referenced parent POM using the `relativePath` (or defaults to the parent directory if omitted).
  - Caches results to avoid re-parsing.
- **Artifact ID (`artifactId`)**: Must be present in each `pom.xml`.
- **Dependencies**: Extracts `(groupId, artifactId)` pairs from the `<dependencies>` section.

4. Dependency Analysis

- Builds a map of `(groupId, artifactId)` to `pom_path` for all modules.
- For each `pom.xml`, checks its dependencies against the module map to find intra-project dependencies.
- Excludes self-dependencies (where a module depends on itself).
- Records dependencies as `(importer_artifactId, imported_artifactId)` pairs.

5. DOT Output

- Outputs a directed graph in DOT format, using `artifactId` as node labels for simplicity.

Example Usage

Suppose your directory structure is irregular:

```
myproject/
  app/pom.xml (groupId="com.myapp", artifactId="app", depends on "core")
  libs/core/pom.xml (groupId="com.myapp", artifactId="core")
  tools/util/pom.xml (groupId="com.myapp", artifactId="util", depends on "core")
```

Run the script:

```
python script.py myproject
```

Output:

```
digraph G {
  "app" -> "core";
  "util" -> "core";
}
```

This shows that both `app` and `util` depend on `core`.

Notes and Limitations

- **No Grouping by Directory**: Unlike the original script in the thinking trace, this version doesn't group modules by directory structure since you didn't request it. It uses `artifactId` directly in the graph.
- **Parent Resolution**: Relies on `relativePath` in the `<parent>` section or assumes the parent POM is in the parent directory if omitted. If the parent POM isn't in the collected set, it raises an error.
- **Simple Matching**: Matches dependencies by `groupId` and `artifactId` only, ignoring versions.
- **No Property Substitution**: Doesn't handle Maven properties like `${project.groupId}`.
- **Error Handling**: Skips problematic `pom.xml` files with a warning instead of failing entirely.

This script meets your requirement to "iterate recursively, get all POMs first, and then analyze"without assuming a fixed file structure. Let me know if you need adjustments!