

Search: 

\$:play start



Tutorial

In this tutorial, you will learn how to use *Neo4j Browser* to comprehend program behaviour.

This interface comprises the top bar (the Search box above), where you can search for program entities (functions, variables, and classes) in the model, and the list of visualization frames (not yet shown - these will be visible after you search for an entity), in which you can visualize and inspect the results of the search. A visualization frame can be maximized to fullscreen and closed whenever you want by clicking the icons at the top.

This guide will show you how to:

1. Search for a program entity in the model
2. Pose questions about the shown entities
3. Expand the model to explore the code behaviour



The code you will use in this study implements the game of Chess. We will use parts of that code as examples in this tutorial. Click on the arrows on the sides or bottom of this visualization frame to navigate through the tutorial.



Search: 

\$:play start



Graphical program data

A program comprises entities (e.g., classes, variables, functions) and the relationships between them (e.g., function calls, variable reads, class containment). A graphical model representing such a program includes nodes representing the entities and links indicating the relationships established in the code.

Consider a function named **isPinned** that checks if a chess piece is pinned by temporarily moving the piece and checking if the move results in the king piece being in "check":

```
bool Board::isPinned(int start, int dest) {  
  
    bool side = arr[start]→getSide();  
    Board boardCopy = *this;  
    Move move{start, dest};  
    boardCopy.movePiece(move);  
    if (boardCopy.isInCheck(side)) {  
        return false;  
    }  
    return true;  
}
```

To find the graphical representation of this function in the model, type "**isPinned**" in the Search box at the top bar and click the blue play button to query the model for an entity with that name.

Search: 


\$:play start



Graphical program data

The new frame shows the data returned by the executed query. The sidebar on the right provides an overview of the node labels and relationships types present in the visualization.

You can reposition the node by dragging it around. If you hover or click on any node of the graph, the overview on the sidebar is replaced by the information associated with the selected element. To return to the overview, you need to deselect the clicked entity. You can do that by clicking on the background or the selected entity once.

You can pan and zoom the visualization in the frame if you want. You can pan around the graph view by clicking and dragging the background. You can zoom in and out by clicking on the buttons in the bottom right corner. You can also expand the visualization frame into fullscreen by clicking on the  button on the top right corner.

Search: 

\$:play start



Graphical program data

Neo4j Browser answers questions you may have about the nodes in the model. When you click a node, the interface opens a donut menu with the options of questions one can ask about the node.

For example, consider we are interested in learning what the arguments of the function **isPinned** are. Click the node to open the question menu and click the option "What are the args for this?" to expand the graph and expose the nodes representing the function's arguments.

```
bool Board::isPinned(int start, int dest) {  
    ...  
}
```

You should see the variable nodes **start** and **dest** connected to the function through a *contain* link.

Search: 

\$:play start



Graphical program data

Neo4j Browser answers questions you may have about the nodes in the model. When you click a node, the interface opens a donut menu with the options of questions one can ask about the node.



Suppose you are wondering which functions can be called by **isPinned**. Click the node to open the question menu and click the option "Who can be called by this?" to expand the graph and expose the nodes representing the function being called.

```
bool Board::isPinned(int start, int dest) {  
  
    bool side = arr[start]→getSide();  
    ...  
    ...  
    boardCopy.movePiece(move);  
    if (boardCopy.isInCheck(side)) {  
        ...  
    }  
    ...  
}
```



You should see functions **getSide**, **movePiece**, and **isInCheck**.



Search: 

\$:play start



Graphical program data

Neo4j Browser answers questions you may have about the nodes in the model. When you click a node, the interface opens a donut menu with the options of questions one can ask about the node.

The links between **isPinned** and functions **getSide**, **movePiece**, and **isInCheck** represent direct calls. If we are interested in learning about indirect calls executed by **isPinned** we must asked the same question to the added nodes.

For example, check if **isInCheck** calls any function. Click on the node **isInCheck** and ask the question "Who can be called by this?"

Search: 

\$:play start



Graphical program data

Neo4j Browser answers questions you may have about the nodes in the model. When you click a node, the interface opens a donut menu with the options of questions one can ask about the node.

You should see function **isUnderAttack**. The current state of the model tells us that **isPinned** can indirectly call **isUnderAttack**. Since its call to **isInCheck** can lead to the execution of **isUnderAttack**. Identifying all indirect calls of a function requires the expansion of the call path until there are no new links to expand.

Search: 

\$:play start



Program comprehension questions

Neo4j Browser answers questions you may have about the nodes in the model. When you click on a node, the interface opens a donut menu with the options of questions one can ask about the node.

The models in *Neo4j Browser* include the following types of links:

1. **function1** call **function2**: *function1* calls *function2*
2. **function** write **variable**: *function* assigns data to *variable*
3. **variable** instanceOf **class**: *variable* stores an instance of *class* (or *struct*)
4. **entity1** contain **entity2**: *entity1* contains *entity2* in some form; for instance, a class contain a function
5. **variable1** varWrite **variable2**: *variable1* is used in an assignment to *variable2*
6. **variable1** parWrite **variable2**: *variable1* is an actual parameter whose value is passed to formal parameter *variable2*

Control flow paths in the code are defined as sequences of *call* links. Data flow paths are sequences of variable assignments (*varWrite*), parameter passing (*parWrite*), and function return values (*retWrite*).

Search: 

\$:play start



Program comprehension questions

Neo4j Browser answers

questions you may have about the nodes in the model. When you click on a node, the interface opens a donut menu with the options of questions one can ask about the node.

To illustrate the representation of dataflow links, consider the function **generateMove**:

```
Move Level1::generateMove() const {  
    ...  
    std::vector<int> canMove;  
    ...  
    int rand1 = rand() % canMove.size();  
    ...  
}
```

Search for the node representing the variable **rand1**. Note that an assignment to the variable is executed within the function, which means that clicking on the question button "Which function writes to this?" adds a new node representing the function **generateMove** and a *write*.

Search: 

\$:play start



Program comprehension questions

Neo4j Browser answers

questions you may have about the nodes in the model. When you click on a node, the interface opens a donut menu with the options of questions one can ask about the node.

```
Move Level1::generateMove() const {  
    ...  
    std::vector<int> canMove;  
    ...  
    int rand1 = rand() % canMove.size();  
    ...  
}
```

Similarly, note that vector **canMove** is used in the assignment to **rand1**. Then, clicking on question button "Which variable writes to this variable?" adds the node representing **canMove** and a *varWrite* link.

Search: 

\$:play start



Program comprehension questions

Neo4j Browser answers

questions you may have about the nodes in the model. When you click on a node, the interface opens a donut menu with the options of questions one can ask about the node.

To illustrate the representation of dataflow links through parameter passing, consider the function **kingMoves**:

```
vector<int> Board::kingMoves(int coord, bool side) {  
    ...  
    if (isUnderAttack(whiteKing, side) != -1) {}  
    ...  
}
```

Search for the node representing the function **isUnderAttack**. In the code, note that function **isUnderAttack** is called and passed two actual parameter **whiteKing** and **side**. To find the formal parameters of the function in the graph, click on the question "What are the args of this?", and add nodes **coord** and **side**.

Search: 

\$:play start



Program comprehension questions

Neo4j Browser answers

questions you may have about the nodes in the model. When you click on a node, the interface opens a donut menu with the options of questions one can ask about the node.

```
vector<int> Board::kingMoves(int coord, bool side) {  
    ...  
    if (isUnderAttack(whiteKing, side) != -1) {}  
    ...  
}
```

For this example, let's focus on the parameter **side**. Select that node and click on the question "Which variable is passed as this argument?". The graph is expanded by adding two nodes also named **side** and two *parWrite* links connecting them to the original **side**. To identify which of the nodes represents the formal parameter **side** declared in the signature of **kingMoves** above, you can ask the question "Where is this declared?" about each of the new nodes. Only one of them should be connected to function **kingMoves** through a *contain* link.

Feel free to proceed to the next slide once you identify the correct node.

Question

Cypher

Search:



\$:play start



Program comprehension questions

Neo4j Browser answers questions you may have about the nodes in the model. When you click on a node, the interface opens a donut menu with the options of questions one can ask about the node.

After posing a question the graph will expand to include the query's results. You can remove the query results by re-clicking on the button of the posed question. Also, clicking on another menu option will remove the results of the previous query and reveal the results of the new query.

If you pose a question that does not change the graph, meaning the question's answer is an empty set, *Neo4j Browser* will not make any changes to the graph (including not removing the results of the previous query). Instead, the yellow box around the question label will turn red to indicate that the results were empty.

Search: 

\$:play start



Program comprehension questions

Neo4j Browser answers

questions you may have about the nodes in the model. When you click on a node, the interface opens a donut menu with the options of questions one can ask about the node.

To experiment with the interface, check which function may call the function **legalSlidingMoves**. Use the question options to identify the name of that function. Move to the next slide to check your answer.

Question

Cypher

Search:



\$:play start



Program comprehension questions

The function that can call *legalSlidingMoves* is named **legalMoves**.

Neo4j Browser answers

questions you may have about the nodes in the model. When you click on a node, the interface opens a donut menu with the options of questions one can ask about the node.

15 / 16



Question

Cypher

Search:



\$:play start



End of tutorial

Now that you are familiar with the interface, we can start the study.

When you are ready, close the visualization frame with results of the tutorial questions (clicking on the 'X' at the right top corner of each frame), and let the researcher know you are ready to proceed with the study.



16 / 16

