Supplement: Case Study: Knight's Tour

For Introduction to Java Programming

By Y. Daniel Liang

This case study can be presented after Chapter 30, "Graphs and Applications."

Graph algorithms are used in everyday life. Delivery companies such as UPS and FedEx use graph algorithms to determine the best delivery routes for their drivers. Google uses graph algorithms to map the best travel routes. The key to solving a real-world problem using graph algorithms is to model the problem into a graph problem. In the text, we model the nine tail problem into the problem of finding a shortest path between two vertices in a graph. This case study introduces a solution of the Knight's Tour problem using graph algorithms.

The Knight's Tour is a well-known classic problem. The objective is to move a knight, starting from any square on a chessboard, to every other square once. Note that the knight makes only L-shaped moves (two spaces in one direction and one space in a perpendicular direction). As shown in Figure 1(a), the knight can move to eight squares.

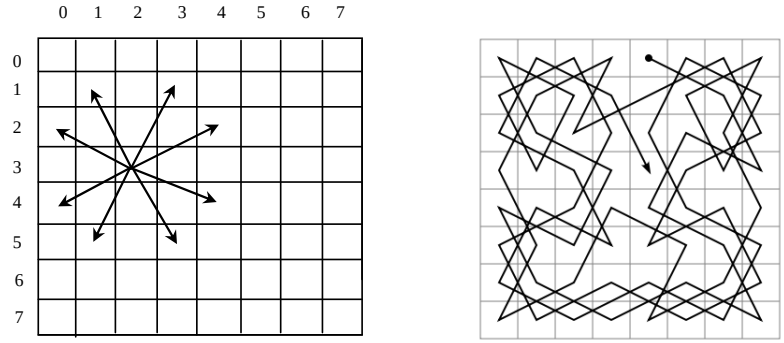

Figure 1

(a) A knight makes an L-shaped move. (b) A solution for a knight to traverse all squares once.

<Side Remark: Hamiltonian path>
<Side Remark: Hamiltonian cycle>

There are several approaches to solving this problem. One approach is to reduce it to a problem for finding a *Hamiltonian path* in a graph—that is, a path that visits each vertex in the graph exactly once. A *Hamiltonian cycle* visits each vertex in the graph exactly once and returns to the starting vertex. To solve the Knight's Tour problem, create a graph with 64 vertices representing all the squares in the chessboard. Two vertices are connected if a knight can move between them.

We will write an applet that lets the user specify a starting square and displays a tour, as shown in Figure 2.

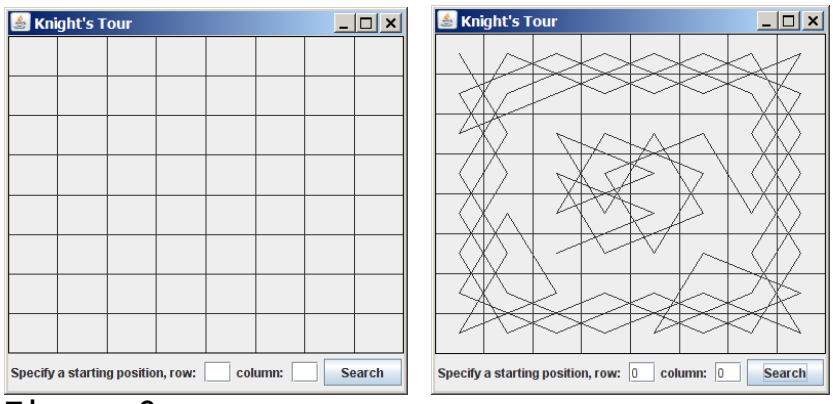

Figure 2

The applet displays a knight's tour.

A graph model was built for the nine tail problem in the preceding section. In the same spirit, we will build a graph model for the Knight's Tour problem. We will create two classes: KnightTourApp and KnightTourModel. The KnightTourApp class is responsible for user interaction and for displaying the solution, and the KnightTourModel class creates a graph for modeling this problem, as shown in Figure 3.

<PD: UML Class Diagram>

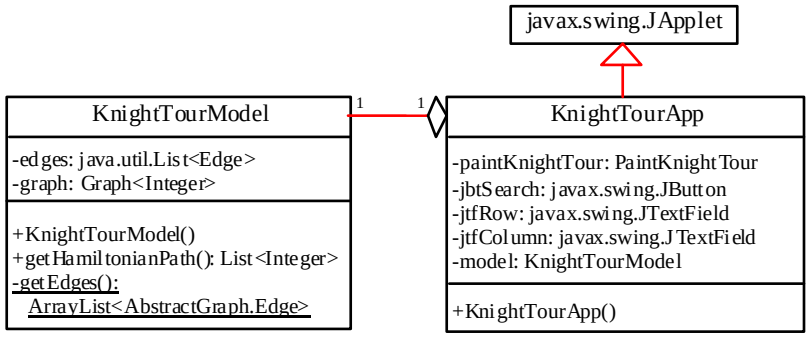

Figure 3

KnightTourModel creates a graph for modeling the problem and KnightTourApp presents a view for the solution.

The KnightTourModel class is given in Listing 1 to create the vertices and the edges. For convenience, the vertices are labeled 0, 1, 2, ..., 63. A square in the chessboard at row i and column j corresponds to the vertex (i * 8 + j). The getEdges() method (lines 20–70) creates all edges and adds them to the edges list. For each node u at square (i, j), check

eight possible edges from u (lines 28–66), as shown in Figure 4. The model builds a graph from the edges and nodes (line 11). The method getHamiltonianPath(int v) returns a Hamiltonian path from the specified starting vertex v by applying the getHamiltonianPath method on the graph.

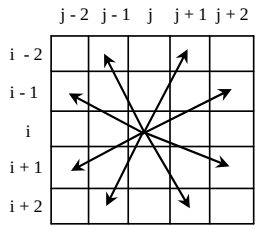

Figure 4

Each node may be connected with eight other vertices.

Listing 1 KnightTourModel.java

PD: Please add line numbers in the following code
***Layout: Please layout exactly. Don't skip the space. This is true for all source code in the book. Thanks, AU.
<Side Remark line 4: edges>
<Side Remark line 6: declare a graph>
<Side Remark line 10: create edges>
<Side Remark line 18: Hamiltonian path>
<Side Remark line 22: create edges>
<Side Remark line 25: vertex label>
<Side Remark line 28: find an edge>

```
import java.util.*;

public class KnightTourModel {
  private UnweightedGraph<Integer> graph; // Define a graph

  public KnightTourModel() {
    // (u, v) is an edge if a knight can move from u and v
    ArrayList<AbstractGraph.Edge> edges = getEdges();

    // Create a graph with 64 vertices labeled 0 to 63
    graph = new UnweightedGraph<Integer>(edges, 64);
  }

  /** Get a Hamiltonian path starting from vertex v */
  public List<Integer> getHamiltonianPath(int v) {
    return graph.getHamiltonianPath(v);
  }

  /** Create edges for the graph */
  public static ArrayList<AbstractGraph.Edge> getEdges() {
```

```
    ArrayList<AbstractGraph.Edge> edges
      = new ArrayList<AbstractGraph.Edge>(); // Store edges
    for (int i = 0; i < 8; i++)
      for (int j = 0; j < 8; j++) {
        int u = i * 8 + j; // The vertex label

        // Check eight possible edges from u
        if (i - 1 >= 0 && j - 2 >= 0) {
          int v1 = (i - 1) * 8 + (j - 2);
          edges.add(new AbstractGraph.Edge(u, v1));
        }

        if (i - 2 >= 0 && j - 1 >= 0) {
          int v2 = (i - 2) * 8 + (j - 1);
          edges.add(new AbstractGraph.Edge(u, v2));
        }

        if (i - 2 >= 0 && j + 1 <= 7) {
          int v3 = (i - 2) * 8 + (j + 1);
          edges.add(new AbstractGraph.Edge(u, v3));
        }

        if (i - 1 >= 0 && j + 2 <= 7) {
          int v4 = (i - 1) * 8 + (j + 2);
          edges.add(new AbstractGraph.Edge(u, v4));
        }

        if (i + 1 <= 7 && j + 2 <= 7) {
          int v5 = (i + 1) * 8 + (j + 2);
          edges.add(new AbstractGraph.Edge(u, v5));
        }

        if (i + 2 <= 7 && j + 1 <= 7) {
          int v6 = (i + 2) * 8 + (j + 1);
          edges.add(new AbstractGraph.Edge(u, v6));
        }

        if (i + 2 <= 7 && j - 1 >= 0) {
          int v7 = (i + 2) * 8 + (j - 1);
          edges.add(new AbstractGraph.Edge(u, v7));
        }

        if (i + 1 <= 7 && j - 2 >= 0) {
          int v8 = (i + 1) * 8 + (j - 2);
          edges.add(new AbstractGraph.Edge(u, v8));
        }
      }

    return edges;
  }
}
```

Now the question is: how can you find a Hamiltonian path in a graph? You can apply the DFS approach to search for a subpath that contains unvisited vertices as deep as possible. In the DFS, after a vertex is visited, it will never be revisited. In the process of finding a Hamiltonian path, you may have to backtrack to revisit the same vertex in order to search for a new path. The algorithm can be described in Listing 2.

Listing 2 Hamiltonian Path Algorithm

PD: Please add line numbers in the following code
***Layout: Please layout exactly. Don't skip the space. This is true for all source code in the book. Thanks, AU.
<Side Remark line 4: visit v>
<Side Remark line 6: all visited?>
<Side Remark line 9: check a neighbor>
<Side Remark line 10: recursive search>

```
/** hamiltonianPath(v) is to find a Hamiltonian path for
 *  all unvisited vertices. */
boolean hamiltonianPath(vertex v) {
  isVisited[v] = true;

  if (all vertices are marked visited)
    return true;

  for each neighbor u of v
    if (u is not visited && hamiltonianPath(u)) {
      next[v] = u; // u is the next vertex in the path from v
      return true;
    }

  isVisited[v] = false; // Backtrack
  return false; // No path starting from v
}
```

To find a Hamiltonian path starting from v, the algorithm first visits v and then recursively finds a Hamiltonian path for the remaining unvisited vertices starting from a neighbor of v (line 10). If so, a Hamiltonian path is found; otherwise, a Hamiltonian path from v does not exist (line 16).

<Side Remark: time complexity>

Let $T(n)$ denote the time for finding a Hamiltonian path in a graph of n vertices and d be the largest degree among all vertices. Clearly,

$$T(n) = O(d \times T(n-1)) = O(d \times d \times T(n-2)) = O(d^n)$$

<Side Remark: NP-complete>

This is an exponential-time algorithm. Can you develop an algorithm in polynomial time? No. It has been proven that the Hamiltonian path problem is *NP-complete*, meaning that no polynomial-time algorithm for such problems can be found. However, you can apply some heuristic to speed up the search. Intuitively, you should attempt to search the

vertices with small degrees and leave those vertices with large degrees open, so there will be a better chance of success at the end of the search.

Add the following two methods in the Graph interface.

```
  /** Return a Hamiltonian path from the specified vertex object
   * Return null if the graph does not contain a Hamiltonian path */
  public java.util.List<Integer> getHamiltonianPath(V vertex);

  /** Return a Hamiltonian path from the specified vertex label
   * Return null if the graph does not contain a Hamiltonian path */
  public java.util.List<Integer> getHamiltonianPath(int index);
```

Implement these methods in the AbstractGraph class as follows:

PD: Please add line numbers in the following code
***Layout: Please layout exactly. Don't skip the space. This is true for all source code in the book. Thanks, AU.
<Side Remark line 26: reorder adjacency list>
<Side Remark line 28: recursive search>

```
  /** Return a Hamiltonian path from the specified vertex object
   * Return null if the graph does not contain a Hamiltonian path */
  public List<Integer> getHamiltonianPath(V vertex) {
    return getHamiltonianPath(getIndex(vertex));
  }

  /** Return a Hamiltonian path from the specified vertex label
   * Return null if the graph does not contain a Hamiltonian path */
  public List<Integer> getHamiltonianPath(int v) {
    // A path starts from v. (i, next[i]) represents an edge in
    // the path. isVisited[i] tracks whether i is currently in the
    // path.
    int[] next = new int[getSize()];
    for (int i = 0; i < next.length; i++)
      next[i] = -1; // Indicate no subpath from i is found yet

    boolean[] isVisited = new boolean[getSize()];

    // The vertices in the Hamiltonian path are stored in result
    List<Integer> result = null;

    // To speed up search, reorder the adjacency list for each
    // vertex so that the vertices in the list are in increasing
    // order of their degrees
    for (int i = 0; i < getSize(); i++)
      reorderNeigborsBasedOnDegree(neighbors.get(i));

    if (getHamiltonianPath(v, next, isVisited)) {
      result = new ArrayList<Integer>(); // Create a list for path
      int vertex = v; // Starting from v
      while (vertex != -1) {
        result.add(vertex); // Add vertex to the result list
        vertex = next[vertex]; // Get the next vertex in the path
      }
    }

    return result; // return null if no Hamiltonian path is found
  }

  /** Reorder the adjacency list in increasing order of degrees */
  private void reorderNeigborsBasedOnDegree(List<Integer> list) {
    for (int i = list.size() - 1; i >= 1; i--) {
      // Find the maximum in the list[0..i]
```

```
        int currentMaxDegree = getDegree(list.get(0));
        int currentMaxIndex = 0;

        for (int j = 1; j <= i; j++) {
          if (currentMaxDegree < getDegree(list.get(j))) {
            currentMaxDegree = getDegree(list.get(j));
            currentMaxIndex = j;
          }
        }

        // Swap list[i] with list[currentMaxIndex] if necessary;
        if (currentMaxIndex != i) {
          int temp = list.get(currentMaxIndex);
          list.set(currentMaxIndex, list.get(i));
          list.set(i, temp);
        }
      }
    }

    /** Return true if all elements in array isVisited are true */
    private boolean allVisited(boolean[] isVisited) {
      boolean result = true;

      for (int i = 0; i < getSize(); i++)
        result = result && isVisited[i];

      return result;
    }

    /** Search for a Hamiltonian path from v */
    private boolean getHamiltonianPath(int v, int[] next,
        boolean[] isVisited) {
      isVisited[v] = true; // Mark vertex v visited

      if (allVisited(isVisited))
        return true; // The path now includes all vertices, thus found

      for (int i = 0; i < neighbors.get(v).size(); i++) {
        int u = neighbors.get(v).get(i);
        if (!isVisited[u] &&
            getHamiltonianPath(u, next, isVisited)) {
          next[v] = u; // Edge (v, u) is in the path
          return true;
        }
      }

      isVisited[v] = false; // Backtrack, v is marked unvisited now
      return false; // No Hamiltonian path exists from vertex v
  }
```

The getHamiltonianPath(V vertex) finds a Hamiltonian path for the specified vertex object, and the getHamiltonianPath(int index) finds a Hamiltonian path for the specified vertex index. getIndex(vertex) returns the label for the vertex object (line 4).

To implement the getHamiltonianPath(int v) method, first create an array next, which keeps track of the next vertex in the path (line 13). next[v] denotes the next vertex in the path after v. Initially, next[i] is set to -1 for all vertices i (line 15). The isVisited array keeps track of the vertices that are currently in the path (line 17). Initially, all elements in isVisited are false.

<Side Remark: heuristics>

To speed up search, the adjacency list for every vertex is reordered in increasing order of their degrees (line 26). So, the vertices with small degrees will be visited earlier.

The recursive getHamiltonianPath(v, next, isVisited) method is invoked to find a Hamiltonian path starting from v. When a vertex v is visited, isVisited[v] is set to true (line 79). If all the vertices are visited, the search returns true (line 82). For each neighbor of v, recursively search a Hamiltonian subpath starting from a neighbor of v (line 85). If a subpath is found, set u to next[v] and return true (line 89). After all neighbors of v are searched without success, backtrack by setting v unvisited (line 93) and return false to indicate a path or a subpath is not found (line 94).

The KnightTourApp class is given in Listing 3 to create the GUI. An instance of the KnightTourModel class is created in line 7. The path of the knight's tour is displayed in an instance of PaintKnightTour panel (line 9). The user specifies the row and column of the starting square and clicks the *Search* button to display the path (line 24).

Listing 3 KnightTourApp.java

PD: Please add line numbers in the following code
***Layout: Please layout exactly. Don't skip the space.
This is true for all source code in the book. Thanks, AU.
<Side Remark line 7: model>
<Side Remark line 8: paint panel>
<Side Remark line 23: button listener>
<Side Remark line 27: display path>
<Side Remark line 28: find path>
<Side Remark line 66: display path>
<Side Remark line 73: main method omitted>

```
import java.util.List;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class KnightTourApp extends JApplet {
  private KnightTourModel model = new KnightTourModel();
  private PaintKnightTour paintKnightTour = new PaintKnightTour();
  private JTextField jtfRow = new JTextField(2);
  private JTextField jtfColumn = new JTextField(2);
  private JButton jbtSearch = new JButton("Search");

  public KnightTourApp() {
    JPanel panel = new JPanel();
    panel.add(new JLabel("Specify a starting position, row: "));
    panel.add(jtfRow);
    panel.add(new JLabel("column: "));
    panel.add(jtfColumn);
    panel.add(jbtSearch);
    add(paintKnightTour, BorderLayout.CENTER);
    add(panel, BorderLayout.SOUTH);

    jbtSearch.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int position = Integer.parseInt(jtfRow.getText()) * 8 +
          Integer.parseInt(jtfColumn.getText());
```

```
        paintKnightTour.displayPath(
          model.getHamiltonianPath(position));
      }
    });
  }

  /** A panel to paint the chessboard and the knight's tour */
  private static class PaintKnightTour extends JPanel {
    private List<Integer> path; // A knight's tour path

    public PaintKnightTour() {
      setBorder(BorderFactory.createLineBorder(Color.black, 1));
    }

    public void displayPath(List<Integer> path) {
      this.path = path;
      repaint();
    }

    protected void paintComponent(Graphics g) {
      super.paintComponent(g);

      // Display horizontal lines
      for (int i = 0; i < 8; i++)
        g.drawLine(0, i * getHeight() / 8,
          getWidth(), i * getHeight() / 8);

      // Display vertical lines
      for (int i = 0; i < 8; i++)
        g.drawLine(i * getWidth() / 8, 0,
          (int)i * getWidth() / 8, getHeight());

      if (path == null) return; // No path to be displayed yet

      for (int i = 0; i < path.size() - 1; i++) {
        int u = path.get(i);
        int v = path.get(i + 1);

        // Knight moves from u and v. Draw a line to connect u and v
        g.drawLine((u % 8) * getWidth() / 8 + getWidth() / 16,
          (u / 8) * getHeight() / 8 + getHeight() / 16,
          (v % 8) * getWidth() / 8 + getWidth() / 16,
          (v / 8) * getHeight() / 8 + getHeight() / 16);
      }
    }
  }
}
```