Hamiltonian Cycle Problem in PDDL

A Detailed Deliverable Package

Idriss Mortadi

Contents

1 Introduction

The Hamiltonian Cycle problem is a classic problem in graph theory and computer science. It involves finding a cycle in an undirected graph that visits each vertex exactly once and returns to the starting vertex. This problem is known to be NP-complete, making it a significant challenge for automated planning and optimization algorithms. In this document, we model the Hamiltonian Cycle problem using the Planning Domain Definition Language (PDDL). We provide a detailed description of the problem, our modeling choices, an illustrative example, and a sample plan to demonstrate the solution. The document is structured to guide the reader through the problem setup, the PDDL model, and the solution process.

2 Problem Description

The objective is to model the Hamiltonian Cycle problem using PDDL, where:

- A **Hamiltonian cycle** is a cycle in an undirected graph that visits each vertex exactly once and returns to the starting vertex.
- The input is an undirected graph $G = (V, E)$ with a set of vertices V and edges E.
- The goal is to find a cycle that visits all vertices exactly once and returns to the starting point, if such a cycle exists.

The model includes three main actions:

1. **Select-Start Action:** Chooses a starting vertex for the cycle.
2. **Move-to-Next Action:** Selects the next unvisited vertex to add to the path.
3. **Complete-Cycle Action:** Completes the cycle by returning to the starting vertex after all vertices have been visited.

3 Modeling Choices and High-Level Explanation

3.1 Type Definitions

We define two types to represent the entities in our problem:

- `vertex`: Represents the nodes in the graph.
- `count`: Represents counters to track the path length.

3.2 Predicates

The following predicates capture the graph structure and the state of the search:

- `(connected ?v1 ?v2 - vertex)`: Indicates there is an edge between vertices v1 and v2.
- `(visited ?v - vertex)`: Indicates vertex v has been included in the cycle.
- `(current ?v - vertex)`: Indicates the current position in the path construction.
- `(start ?v - vertex)`: Indicates the starting vertex of the cycle.
- `(path-length ?n - count)`: Tracks the number of vertices visited.
- `(next ?n1 ?n2 - count)`: Defines the successor relation for counts.
- `(total-vertices ?n - count)`: Stores the total number of vertices in the graph.

3.3 Actions

Select-Start Action: This action selects the starting vertex for the Hamiltonian cycle. Its parameters include the vertex to select and count objects to track path length. The preconditions ensure that:

- No start vertex has been selected yet.
- The path length is zero.

Its effects establish the starting vertex, mark it as visited, and set the path length to one.

Move-to-Next Action: This action extends the path by moving from the current vertex to a connected, unvisited vertex. Its parameters include the current vertex, the next vertex to visit, and count objects. The preconditions ensure that:

- The agent is at the "from" vertex.
- There is an edge connecting the "from" and "to" vertices.
- The "to" vertex has not been visited yet.
- The path length is tracked correctly.

Its effects update the current position, mark the new vertex as visited, and increment the path length.

Complete-Cycle Action: This action completes the Hamiltonian cycle by connecting the last vertex back to the starting vertex. Its parameters include the last vertex, the first vertex, and the path length. The preconditions ensure that:

- The agent is at the last vertex.
- There is an edge connecting the last vertex to the first vertex.
- All vertices have been visited (path length equals total number of vertices).

Its effects move the current position back to the starting vertex, completing the cycle.

4 Illustrative Instance

The following is an example instance from our implementation:

4.1 Instance Setup

- The example uses a graph with 5 vertices, labeled v1 through v5.
- Edges are created with a probability of 0.4, ensuring the graph is connected.
- The graph is represented as an undirected graph, so each connection is bidirectional.
- Count objects n0 through n5 are used to track the path length.

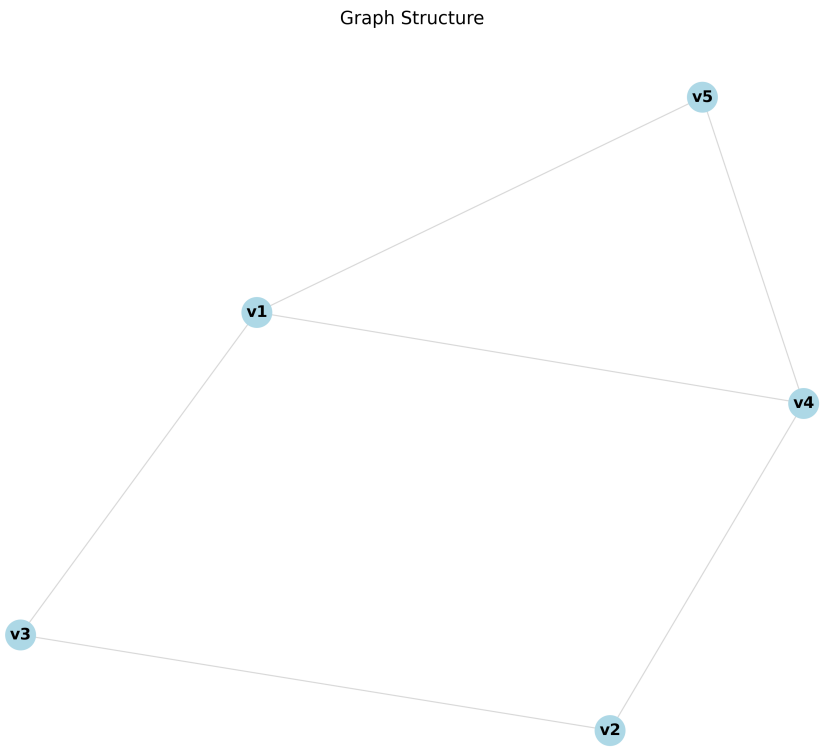

Figure 1: The original graph structure before finding a Hamiltonian cycle. Vertices are labeled v1 through v5, and edges represent valid connections between vertices.

4.2 Step-by-Step Example

Let's consider a small part of the execution for illustration:

1. **Initial Configuration:**
   - All vertices are unvisited.
   - No starting vertex is selected.
   - Path length is zero.
2. **Step 1:** Execute `select-start` action with parameters:

   ```
   (select-start v5 n0 n1)
   ```

   Effects:
   - v5 becomes the starting vertex.
   - v5 is marked as visited.
   - Current position is set to v5.
   - Path length becomes 1.
3. **Step 2:** Execute `move-to-next` action:

   ```
   (move-to-next v5 v4 n1 n2)
   ```

   Effects:

- Current position moves from v5 to v4.
- v4 is marked as visited.
- Path length becomes 2.

4. **Continuing Steps:** The process continues, with each step selecting an unvisited vertex adjacent to the current one and extending the path.
5. **Final Step:** After all vertices have been visited (path length is 5), execute `complete-cycle` action:

   ```
   (complete-cycle v1 v5 n5)
   ```

   This connects the last vertex (v1) back to the starting vertex (v5), completing the Hamiltonian cycle.

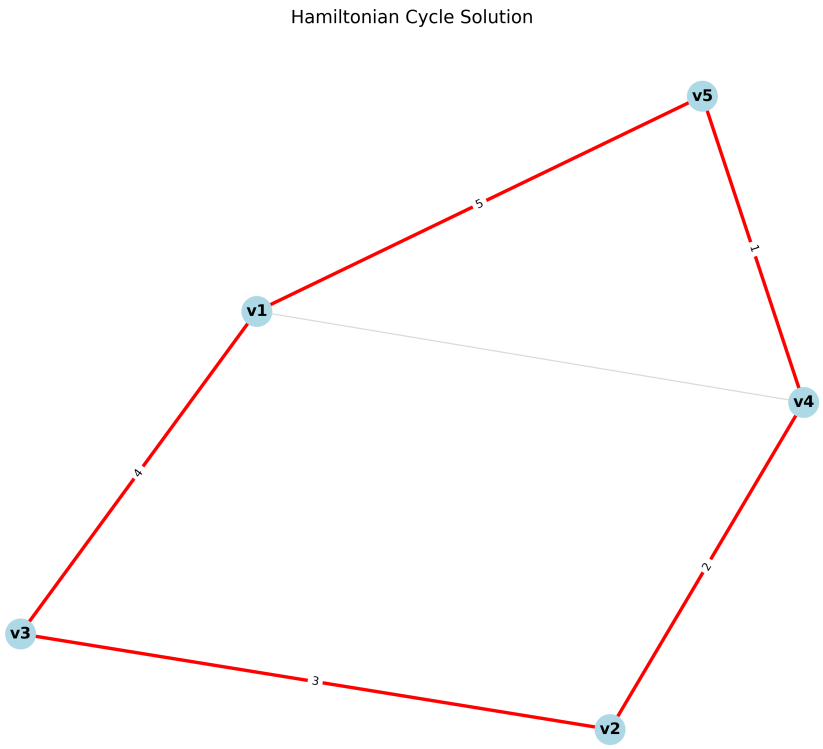

Figure 2: The graph with the computed Hamiltonian cycle solution highlighted in red. The numbers on the edges indicate the order in which vertices are visited in the cycle.

5 Sample Plan

The following is a sample plan for the Hamiltonian Cycle problem instance described above:

(select-start v5 n0 n1)
(move-to-next v5 v4 n1 n2)
(move-to-next v4 v2 n2 n3)
(move-to-next v2 v3 n3 n4)
(move-to-next v3 v1 n4 n5)
(complete-cycle v1 v5 n5)
(reach-goal)

6 Justification and Challenges

6.1 Why This Problem Is Interesting

- **NP-Completeness:** The Hamiltonian cycle problem is NP-complete, making it an excellent test case for evaluating planning algorithms' ability to handle computationally challenging problems.
- **Real-World Applications:** The problem has applications in logistics, circuit design, and network routing, making it practically relevant.
- **Search Space Complexity:** The problem exemplifies the combinatorial explosion that planners face, as the number of possible paths grows factorially with the number of vertices.

6.2 Challenges Posed by the Problem

- **Exponential Solution Space:** For a graph with n vertices, there are potentially (n-1)! different cycles to consider, creating a vast search space for the planner.
- **Graph Connectivity:** The solution depends critically on the graph's structure, requiring careful consideration of edge connectivity.
- **Performance Scaling:** As the number of vertices increases, the planner's performance can degrade rapidly, testing the limits of current planning algorithms.

7 Graph Generation Script

We provide a Python script that generates random connected graphs and creates corresponding PDDL problem files. The script uses NetworkX for graph generation and Matplotlib for visualization. To use the script, run:

```
python hamiltonian_cycle.py --vertices <num_vertices> --edge-prob <probability>
```

This will generate a graph, create a PDDL problem file, solve the problem, and create visualizations of the original graph and the solution.

Installation

To install the required Python packages, run:

```
pip install networkx matplotlib planutils
```

FF Planner

The script requires the FF planner to be accessible in your system's PATH. You can install the FF planner using the planutils Python package by running:

```
planutils install ff
```