

GitHub reference sources:

<https://github.com/zobront/dna-computing-simulator/tree/master>

<https://github.com/johncburns1/DNAComputationSim>

This Python code (Link 1) simulates Leonard Adleman's DNA computing experiment to solve a Hamiltonian Path problem. Let me break down the steps in detail to explain how the computation is performed:

---

## Background

In a Hamiltonian Path problem, the goal is to find a path in a graph that visits every node exactly once and ends at a specific node. Adleman used DNA strands to encode nodes and edges and applied biological processes to filter solutions systematically.

## Steps in the Code

---

### 1. Vertex and Edge DNA Strand Creation

- **Vertex strands** represent graph nodes, and each is a unique DNA sequence.
- **Edge strands** represent connections between nodes, created by concatenating partial matches of vertex strands.
- **Complements** are the reverse complement sequences needed to form bridges between DNA strands.

**Purpose:** This encodes the graph structure into DNA sequences.

---

### 2. Random Path Creation

The code simulates the mixing of DNA to form random paths by:

1. Multiplying the edge list (representing connections) 10,000 times to mimic the abundance of DNA molecules.
  2. Randomly shuffling the edges to simulate random connections.
  3. Constructing paths by matching complementary ends of strands.
- If a growing strand finds a match with an edge, it appends that edge to grow the path.
  - Paths ending at the final node are saved.

**Output:** The total number of random paths formed.

---

### 3. Filtering Paths with Correct Start and End

- Using **Polymerase Chain Reaction (PCR)** in the real experiment, only paths that start at the source node and end at the target node are amplified.
- In the code, paths are filtered to keep only those with the correct start and end sequences.

**Output:** The number of paths with valid start and end nodes.

---

#### 4. Filtering Paths by Length

- Using **Gel Electrophoresis**, DNA strands of a specific length (corresponding to NN vertices) are isolated.
- The code calculates valid paths by ensuring their length matches the number of nodes times a fixed length (10 bases per node).

**Output:** The number of paths with the correct number of steps.

---

#### 5. Filtering Paths Including All Nodes

- Each node must appear at least once in the final path. In the real experiment, this is done by adding primers for each node's strand.
- The code iteratively filters paths that do not contain all nodes.

**Output:** The remaining paths that visit all nodes exactly once.

---

#### 6. Decoding the Solution

- The final remaining path(s) are decoded to extract the sequence of nodes in the Hamiltonian Path.
  - The output in the code is a solution path: ['I', 'L', 'O', 'V', 'E', 'D', 'N', 'A'].
- 

### Key Results

- **Random paths created:** Simulated random DNA paths.
  - **Paths with correct start and end:** Focused on viable candidates for solutions.
  - **Paths with NN vertices:** Ensured paths touch all nodes.
  - **Paths including all nodes:** Guaranteed the path is a valid Hamiltonian Path.
- 

### How Computing Happens

1. **Encoding Information:** DNA sequences encode the problem (nodes and edges).
2. **Massive Parallelism:** The code simulates DNA's ability to process many solutions simultaneously by generating random paths.

3. **Filtering:** Biological processes like PCR and gel electrophoresis are simulated as filters to systematically eliminate invalid paths.
  4. **Decoding the Solution:** The valid DNA strand corresponds to the Hamiltonian Path.
- 

Let me know if you want further details on any specific step!

You said:

Thanks. Can you draw the text based block diagram with the functions and process used and any intermediate inputs, outputs and key steps

ChatGPT said:

ChatGPT

Here's a text-based block diagram that outlines the functions, processes, and key intermediate inputs and outputs for the DNA computing simulation:

---

```
[Start]
↓
1. Vertex and Edge Strand Creation (Input: Nodes and Edges)
  - Function: Generate unique DNA sequences for each vertex and edge.
  - Output:
    * Vertex Strands: ['TAAAGCGTCC', 'TGTAACCTGA', ...]
    * Edge Strands: ['TAAAGCGTCCTGTAA', 'CCTGAAGTTC', ...]
    * Complements: ['ATTCGCGAGG', 'TCAAGGCAAG', ...]

↓
2. Random Path Creation (Input: Edge Strands, Complements)
  - Function: Simulate DNA mixing and strand growth.
  - Key Steps:
    * Shuffle edge strands (simulate random paths).
    * Match complements to grow valid strands.
  - Output:
    * Path Strands (Random Paths): ['...']

↓
3. Filter by Start & End Nodes (Input: Path Strands)
  - Function: Simulate PCR to keep paths starting and ending at correct nodes.
  - Filtering Criterion:
    * Start: Match first vertex DNA sequence.
    * End: Match last vertex DNA sequence.
  - Output:
    * Valid Start-End Strands: ['...']

↓
4. Filter by Length (Input: Valid Start-End Strands)
  - Function: Simulate Gel Electrophoresis to isolate paths of correct length.
  - Filtering Criterion:
    * Length:  $(10 \times N)$  bases (where  $(N)$  is the number of vertices).
  - Output:
    * Length-Filtered Strands: ['...']

↓
5. Filter for Node Coverage (Input: Length-Filtered Strands)
  - Function: Ensure paths include all vertices at least once.
```

- Filtering Criterion:
  - \* Each vertex DNA sequence appears in the path.
- Output:
  - \* Node-Inclusive Strands: ['...']

↓

6. Decode Solution (Input: Node-Inclusive Strands)
- Function: Map DNA sequences back to graph nodes.
  - Output:
    - \* Hamiltonian Path Solution: ['I', 'L', 'O', 'V', 'E', 'D', 'N', 'A']

↓

[End]

---

## Key Details

### 1. Intermediate Inputs and Outputs:

- Vertex and Edge strands serve as the building blocks for paths.
- Path strands are filtered through several layers to eliminate invalid candidates.
- Final output is a Hamiltonian Path (a sequence of nodes).

### 2. Processes Simulated:

- **Random Path Creation:** DNA mixing and complement matching.
  - **Start-End Filtering:** PCR amplification with primers.
  - **Length Filtering:** Gel Electrophoresis for strand length.
  - **Node Coverage Filtering:** Ensure all nodes are touched at least once.
- 

Let me know if you'd like a visual diagram or further elaboration on any block!

**You said:**

A simple visual diagram can help visualize the steps

**ChatGPT said:**

ChatGPT

Here is a simple visual block diagram illustrating the steps of the DNA computing simulation for solving the Hamiltonian Path problem. Let me know if you'd like further edits or annotations.

**You said:**

can this diagram be more simplified with clear step numbers and how the parallel operations happen, any iterations done

**ChatGPT said:**

ChatGPT

Here is the updated and simplified block diagram. It highlights the key steps, parallel operations, and iterations in the DNA computing simulation for solving the Hamiltonian Path problem. Let me know if additional refinements are needed!

**You said:**

I feel text based block diagram will be easier to see

**ChatGPT said:**

ChatGPT

Here's a simplified **text-based block diagram** that incorporates step numbers, parallel operations, and iterations for clarity:

---

```
[Start]
↓
(1) Vertex and Edge Strand Creation
  - Create DNA sequences for graph nodes (vertices) and connections (edges).
  - Output:
    * Vertex Strands
    * Edge Strands
    * Complement Strands

↓
(2) Random Path Creation (Parallel)
  - Simulate random mixing of edge strands.
  - Iterate over shuffled edges to grow paths using complement matching.
  - Parallel: Thousands of paths form simultaneously.
  - Output:
    * Path Strands (Random Paths)

↓
(3) Start-End Filtering
  - Use PCR to keep paths starting and ending at correct nodes.
  - Iteration: Check each path for valid start and end sequences.
  - Output:
    * Start-End Filtered Strands

↓
(4) Length Filtering
  - Use Gel Electrophoresis to isolate paths with correct length.
  - Iteration: Filter paths with length =  $(10 \times N)$  bases.
  - Output:
    * Length-Filtered Strands

↓
(5) Node Coverage Filtering (Parallel)
  - Ensure paths include all vertices at least once.
  - Parallel: Paths are checked for node inclusion simultaneously.
  - Iteration: Eliminate paths missing any vertex.
  - Output:
    * Node-Inclusive Strands

↓
(6) Decode Solution
  - Decode remaining valid DNA strands into graph nodes.
  - Output:
    * Hamiltonian Path Solution: ['I', 'L', 'O', 'V', 'E', 'D', 'N', 'A']

↓
[End]
```

---

## Key Features

1. **Parallelism:** Highlighted in steps 2 and 5 where multiple paths are processed simultaneously.
2. **Iterations:** Emphasized in steps 3, 4, and 5, where paths are sequentially filtered based on criteria.
3. **Clarity:** Steps are clearly numbered and described with intermediate inputs and outputs.

Let me know if additional adjustments are needed!

You said:

Thanks, this is better now.

ChatGPT said:

ChatGPT

You're welcome! I'm glad this version works better for you. Let me know if you have any other questions or need further assistance! ☐

## CODE:

```
import random

# STEP 0: PREP
# - The experimenters want to create DNA strands that encode for the nodes and edges of a graph
# - We will create a graph with 8 vertices (they used 7), and encode the edges between them with a
list of tuples
# - We will encode each node with a random string of bases 10 bases long (to ensure randomness and
to make molecules stable at room temp)
# - We'll encode edges with the last 5 bases of the vertex at the start of the edge, and the first
5 of the end of the edge
# - Finally, we'll make strands complementary to the vertex strands
# - The result is that edges and complementary strands can connect to form random paths through the
graph

vertex_names = ['I', 'L', 'O', 'V', 'E', 'D', 'N', 'A']
V = len(vertex_names)
edges = [(0, 1), (0, 2), (0, 5), (1, 2), (1, 7), (2, 3), (3, 2), (3, 5), (3, 4), (4, 5), (5, 1),
(5, 6), (6, 7)]

pairings = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
K = 10

vertex_strands = [ ''.join([ random.choice(list(pairings.keys())) for _ in range(K) ]) for _ in
range(V) ]
print(f"Vertex Strands Generated: {vertex_strands}")

def find_edge_strands(vertices, edges):
    edge_strands = [ vertices[v1][(K//2):] + vertices[v2][: (K//2)] for (v1, v2) in edges ]
    return [ strand.replace(vertices[0][-(K//2):], vertices[0]).replace(vertices[-1][: (K//2)],
vertices[-1]) for strand in edge_strands ]

edge_strands = find_edge_strands(vertex_strands, edges)
print(f"\nEdge Strands Created: {edge_strands}")
```

```
def find_complement(strand):
    output = ""
    for nuc in strand: output += pairings[nuc]
    return output

complements = [ find_complement(v) for v in vertex_strands ]
print(f"\nComplements Created: {complements}")
```