Eulerian Tours

By Shaylan Lalloo

What is an Eulerian Tour?

- A path that uses every edge exactly once is called an eularian tour. Furthermore, a path that starts and ends at the same vertex and is an eularian tour but is called an eulerian circuit.

When does there exist an eulerian tour?

- An eulerian tour exists when the degree of all vertices except for exactly 2 are even and the graph is connected
- An eularian circuit exists when the degree of all the vertices are even and the graph is connected

Proof

- This can be seen from the fact that every time you enter a vertex in a path, you must be able to leave it unless you are at the beginning or end of the path so this adds 2 to the degree of the vertices on the path not being the starting or ending vertex.

Algorithm for finding eulerian tours

- Find the starting node. Then recurse using the following rule
  - If a node has no neighbours, push it onto the answer vector
  - If a node has a neighbour, throw the neighbours onto a stack and process them
  - Processing a node consists of deleting the edge between the current node and neighbour, then recursing on the neighbour. Once that is done, pushing the current node onto the answer vector

Code(Variables)

- vector<int> mygraph[10];
- int n;
- vector<int> mystack;
- vector<int> myans;
- int curpos = 0;

Reading Inputs

- ifstream fin ("myin.txt");

```
fin >> n;

for (int i = 0; i < n; ++i){
    int f, t;
    fin >> f >> t;
    mygraph[f - 1].push_back(t - 1);
    mygraph[t - 1].push_back(f - 1);
}
for (int i = 0; i < 7; ++i){
    sort(mygraph[i].begin(), mygraph[i].end(), cmp);
}
```

Recursion algorithm but using stack

```
mystack.push_back(0);
  while (!mystack.empty()){
     curpos = mystack.back();
     if (mygraph[curpos].size() == 0){
        myans.push_back(curpos);
        mystack.pop_back();
     }
     else {
        int neigh = mygraph[curpos].back();
        mystack.push_back(neigh);
        mygraph[curpos].pop_back();
        for (int i = 0; i < mygraph[neigh].size(); ++i){
           if (mygraph[neigh][i] == curpos){
              mygraph[neigh].erase(mygraph[neigh].begin() + i);
              break;
           }
        }
     }
  }
```

Outputting result

- cout << "MYANS: ";
- for (int i = 0; i < myans.size(); ++i){
- cout << myans[i] + 1 << " ";
- }
- cout << endl;

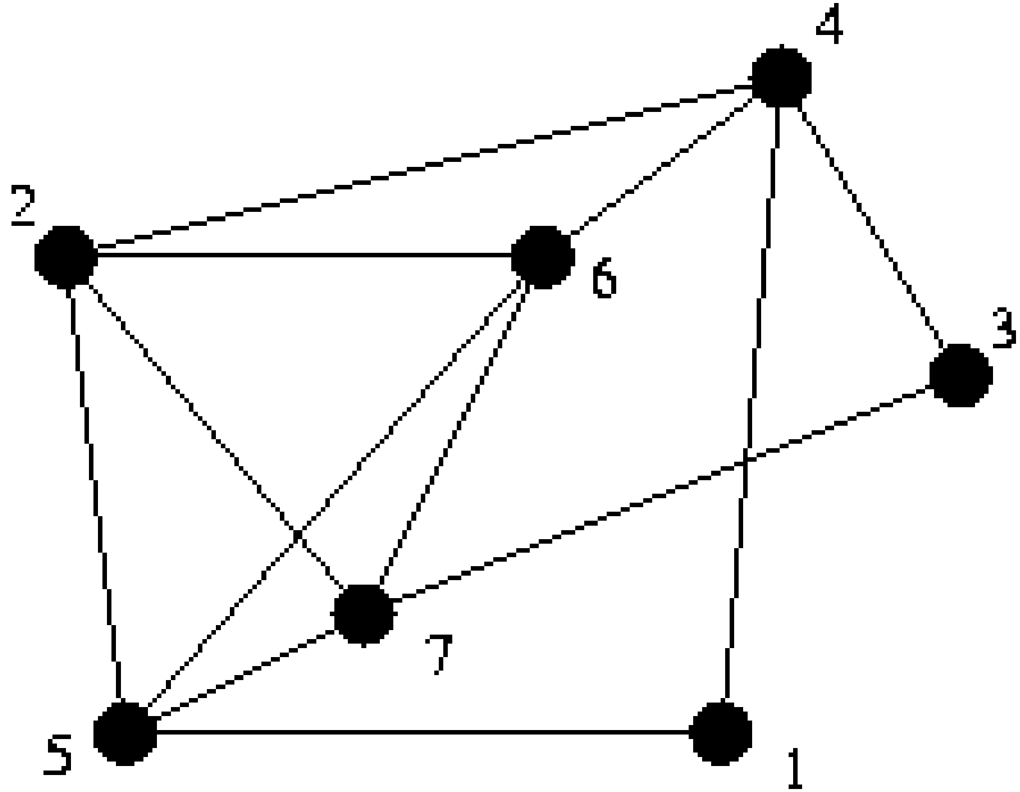
Pseudocode

```
# circuit is a global array
  find_euler_circuit
    circuitpos = 0
    find_circuit(node 1)

# nextnode and visited is a local array
# the path will be found in reverse order
  find_circuit(node i)

    if node i has no neighbors then
      circuit(circuitpos) = node i
      circuitpos = circuitpos + 1
    else
      while (node i has neighbors)
          pick a random neighbor node j of node i
          delete_edges (node j, node i)
          find_circuit (node j)
      circuit(circuitpos) = node i
      circuitpos = circuitpos + 1
```

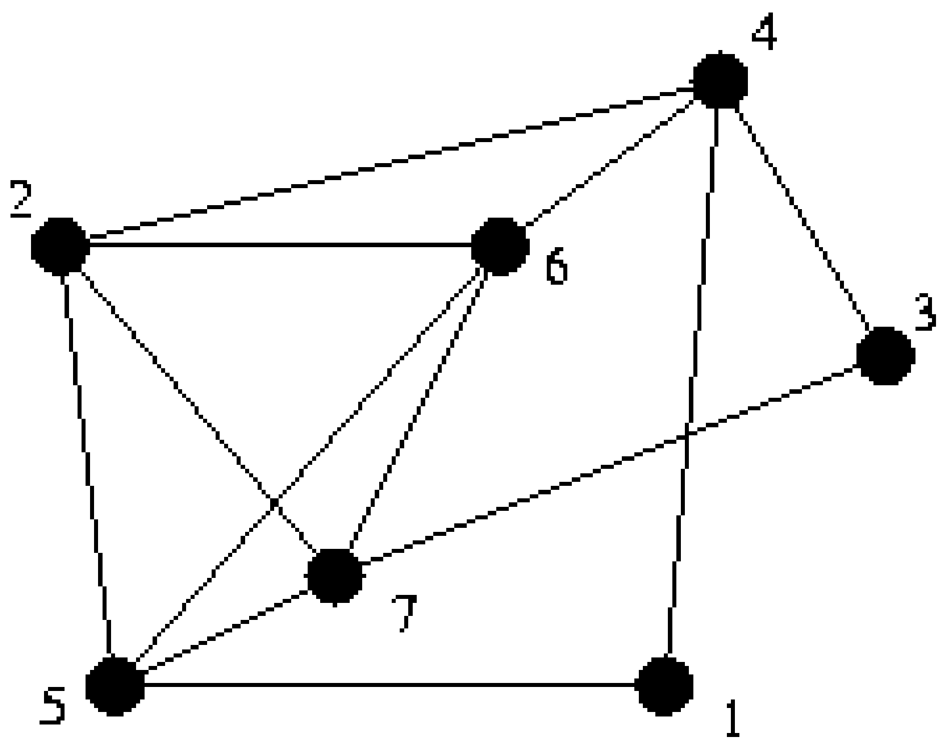
Visual representation of algorithm



Stack:

Location: 1

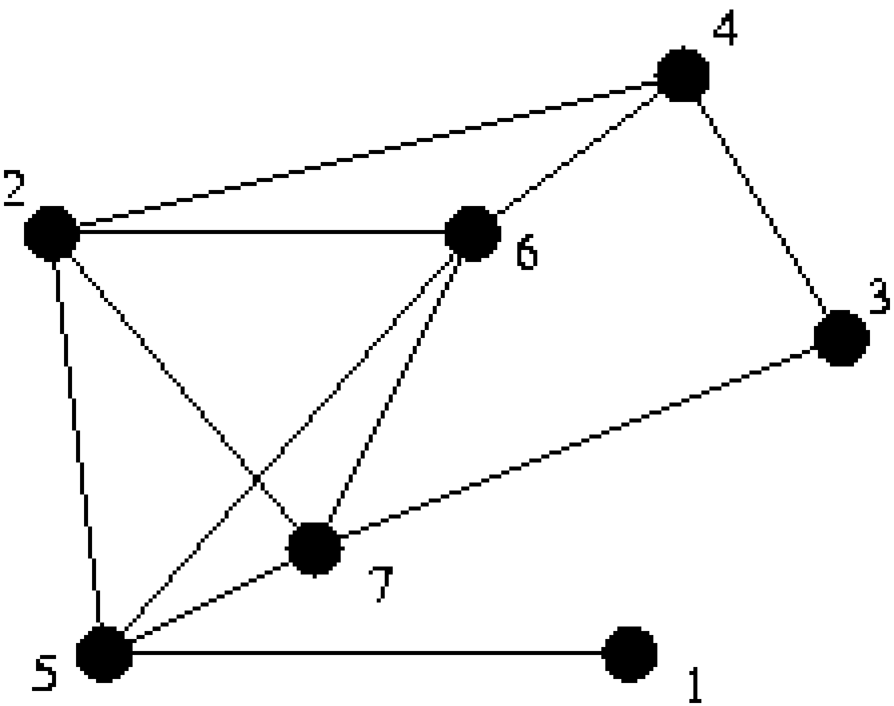
Circuit:



Stack: 1

Location: 4

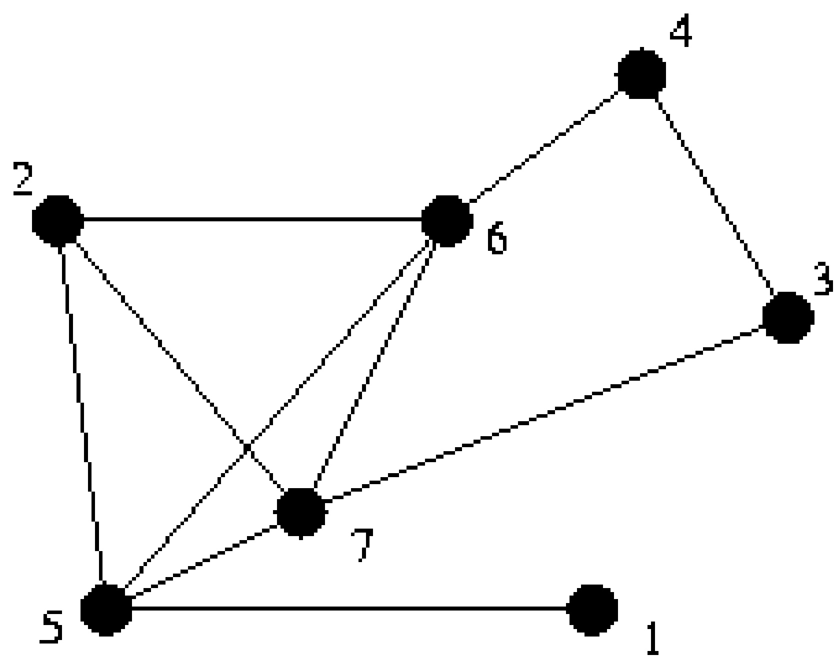
Circuit:



Stack: 1 4

Location: 2

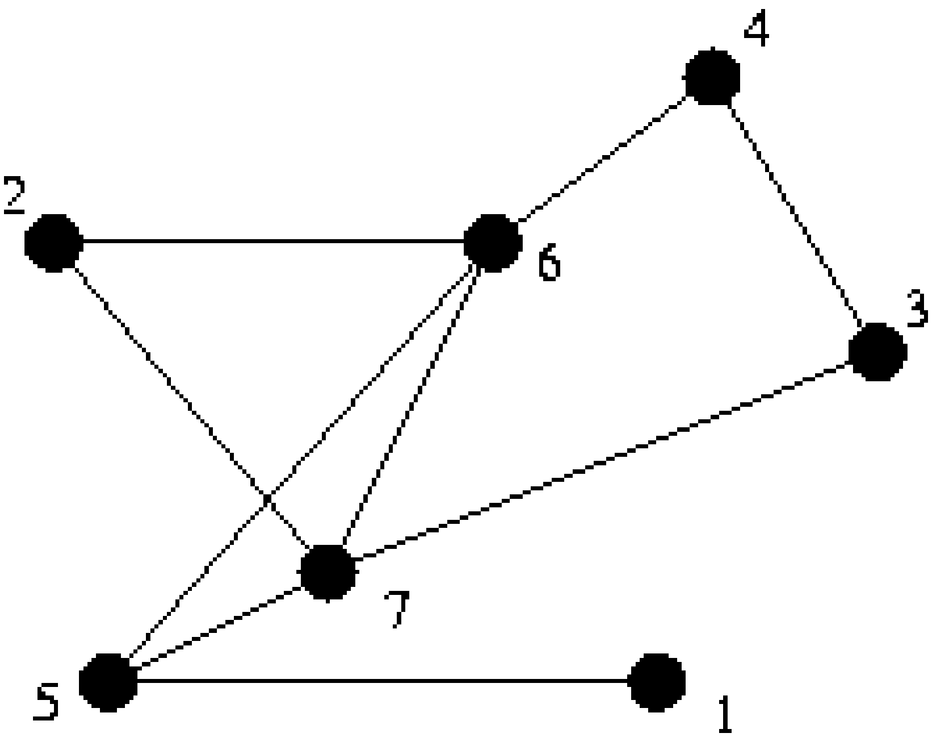
Circuit:



Stack: 1 4 2

Location: 5

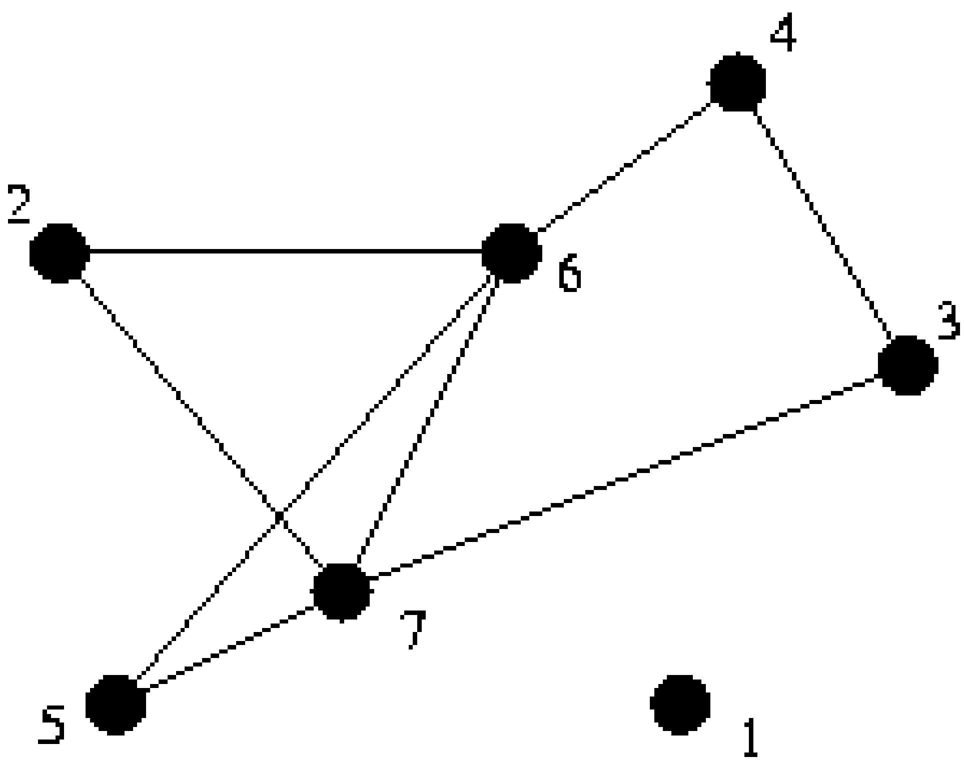
Circuit:



Stack: 1 4 2 5

Location: 1

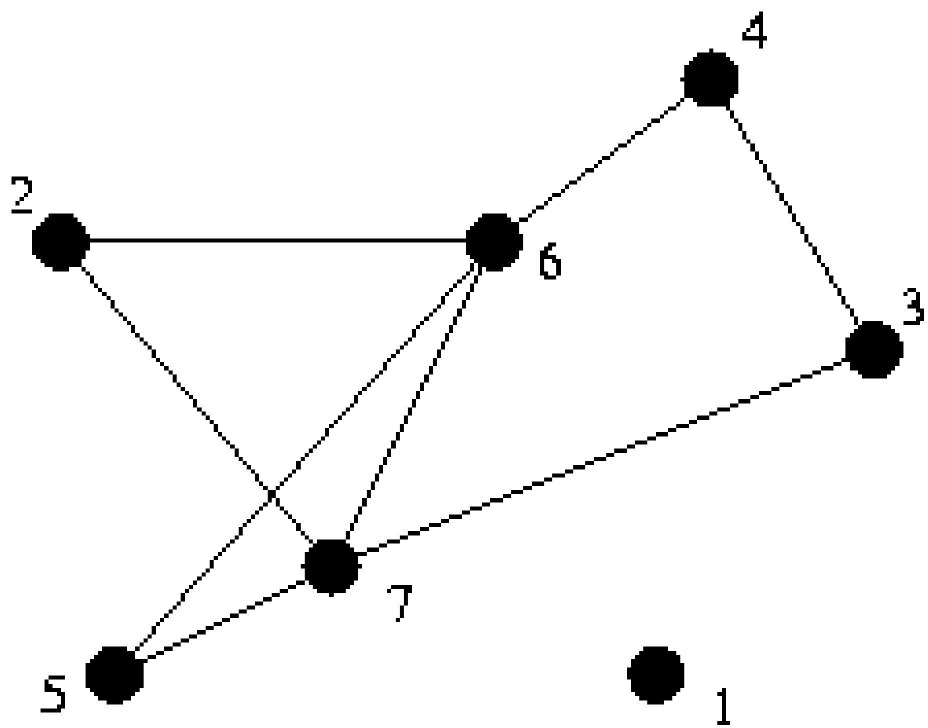
Circuit:



Stack: 1 4 2

Location: 5

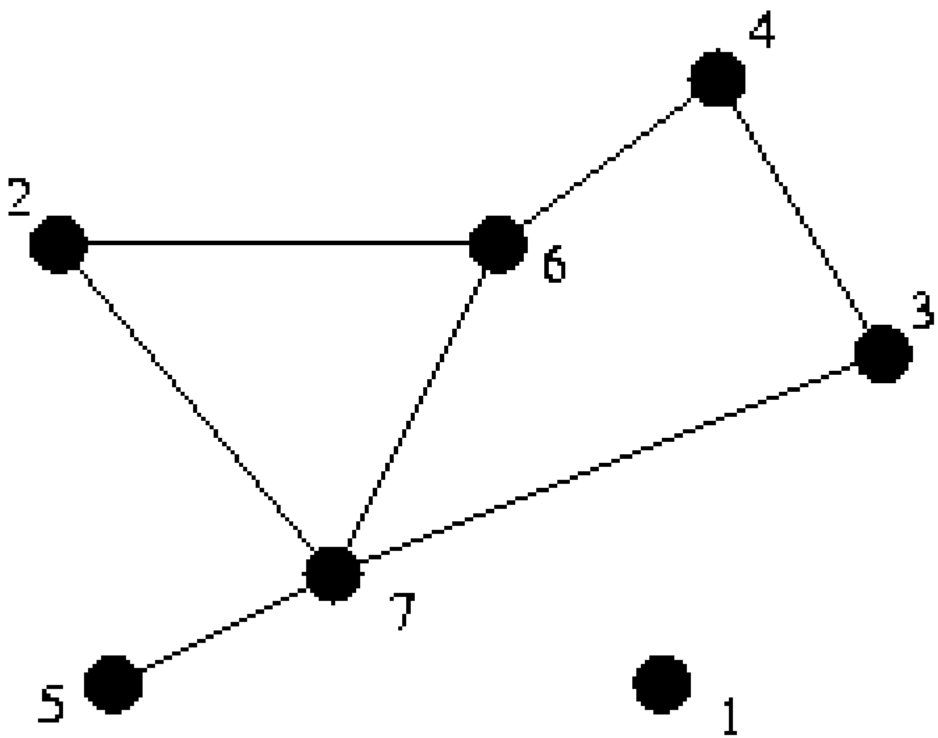
Circuit: 1



Stack: 1 4 2 5

Location: 6

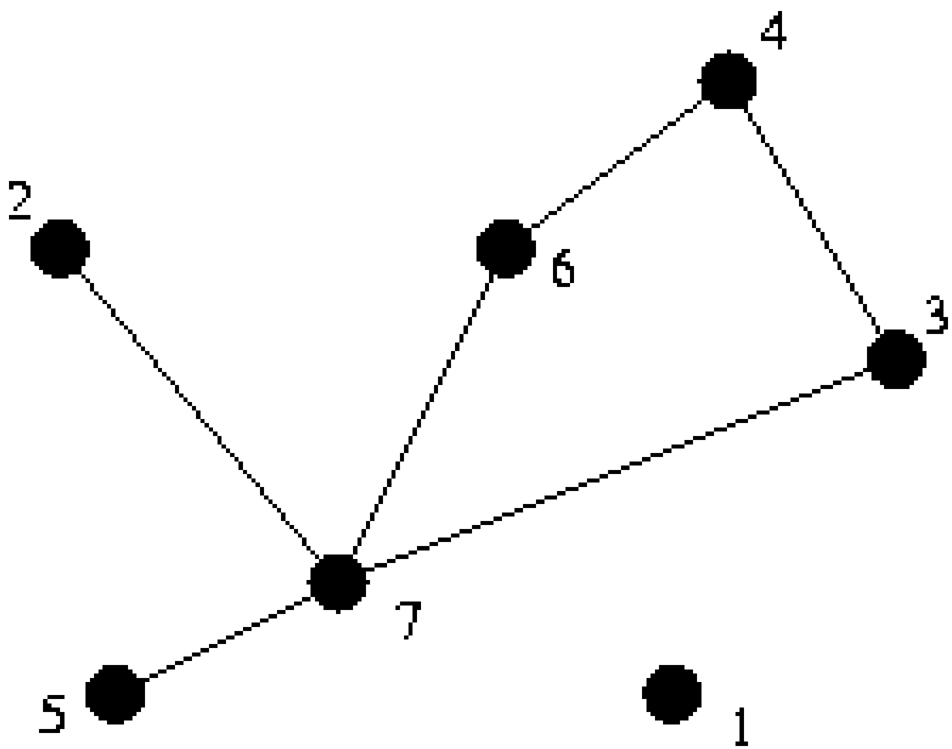
Circuit: 1



Stack: 1 4 2 5 6

Location: 2

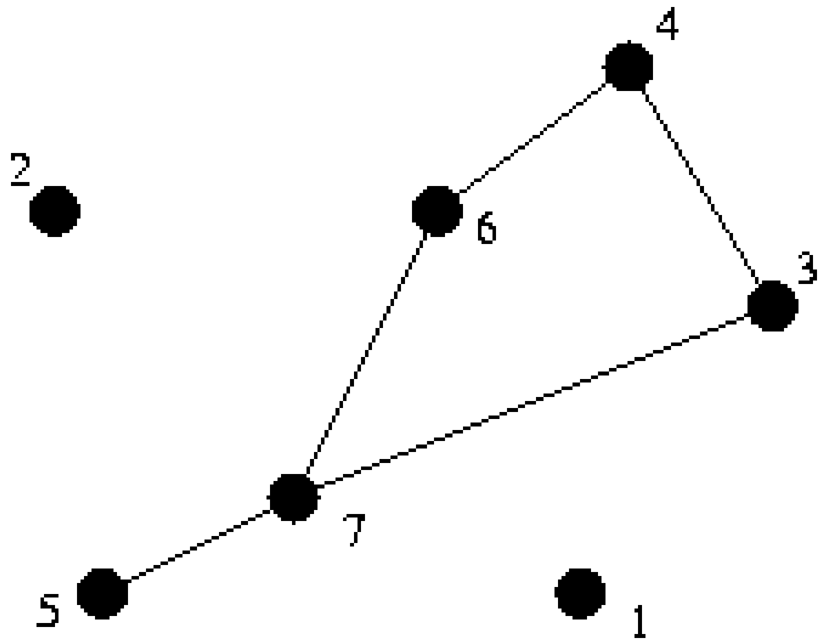
Circuit: 1



Stack: 1 4 2 5 6 2

Location: 7

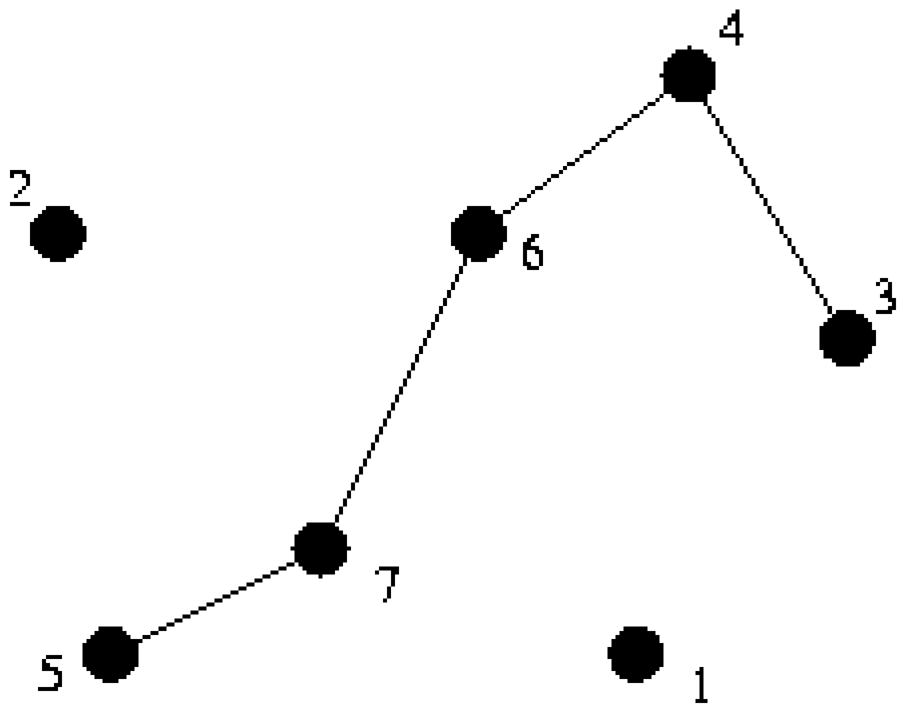
Circuit: 1



Stack: 1 4 2 5 6 2 7

Location: 3

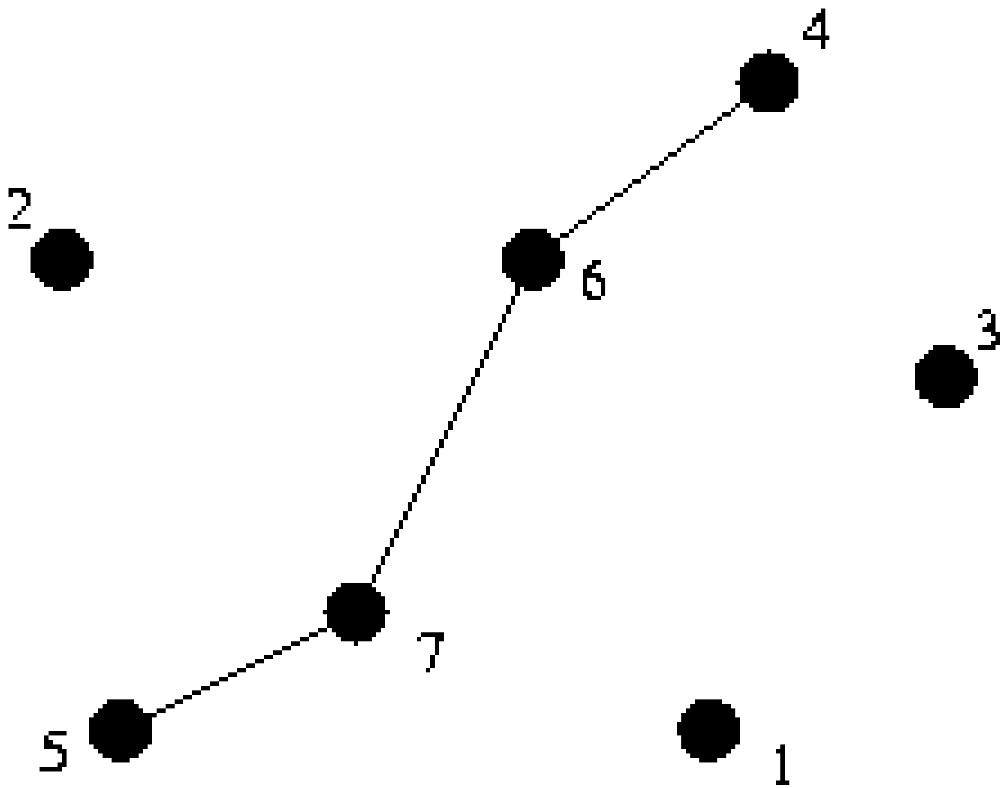
Circuit: 1



Stack: 1 4 2 5 6 2 7 3

Location: 4

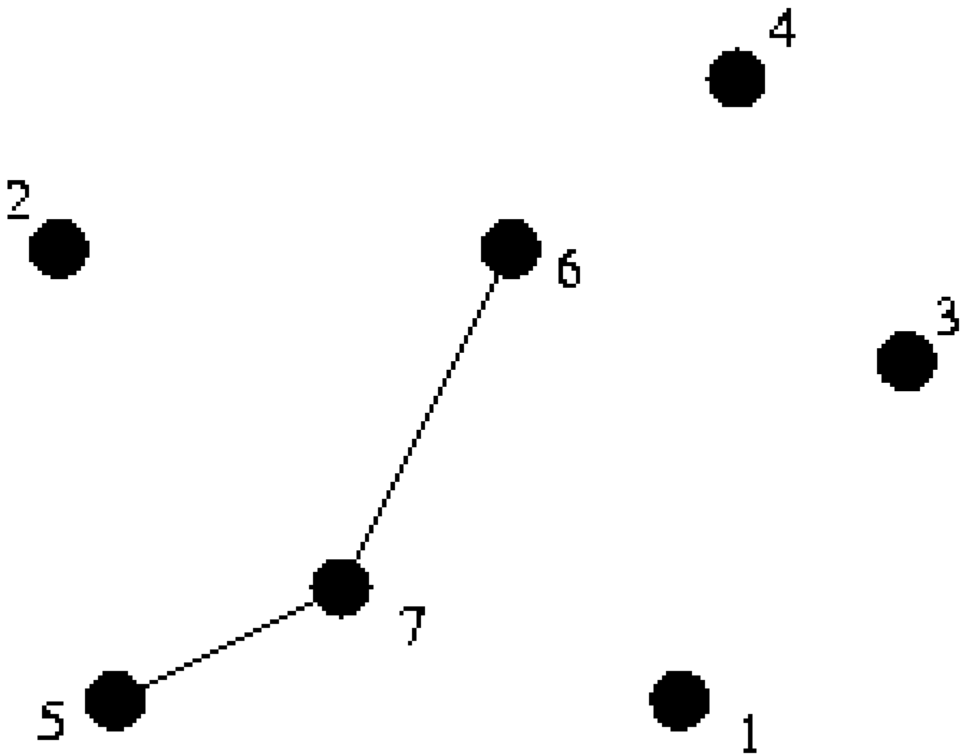
Circuit: 1



Stack: 1 4 2 5 6 2 7 3 4

Location: 6

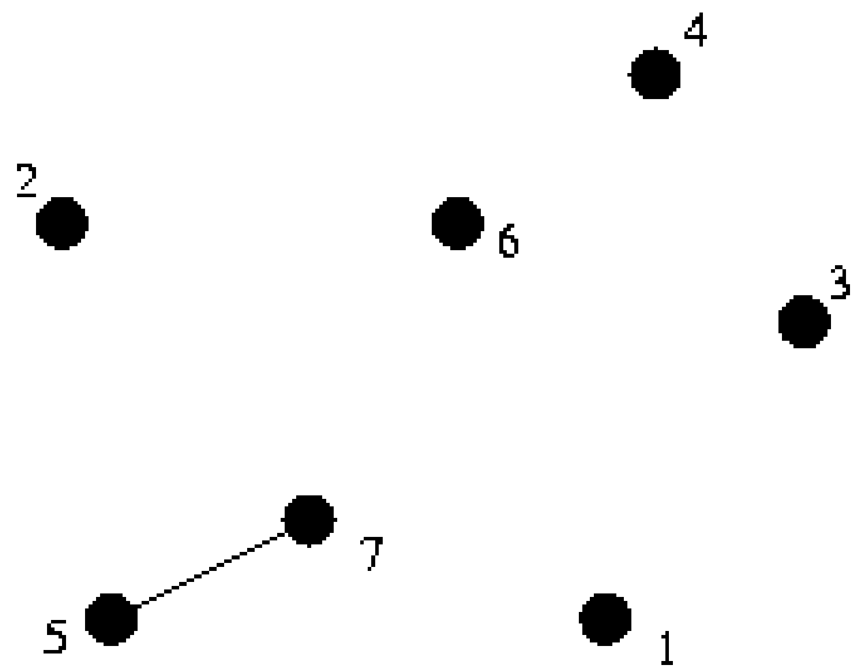
Circuit: 1



Stack: 1 4 2 5 6 2 7 3 4 6

Location: 7

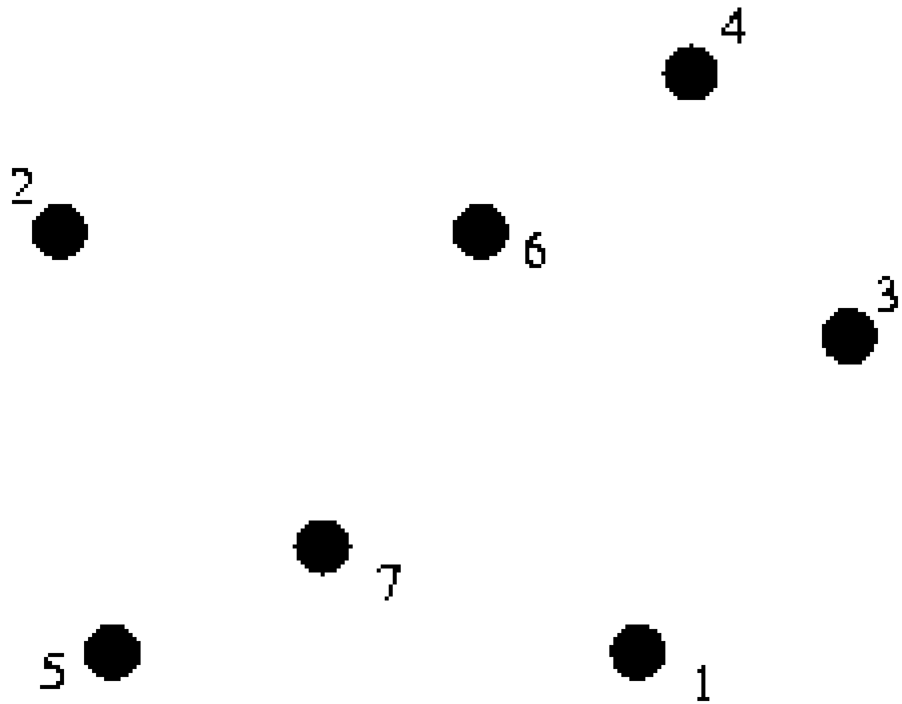
Circuit: 1



Stack: 1 4 2 5 6 2 7 3 4 6 7

Location: 5

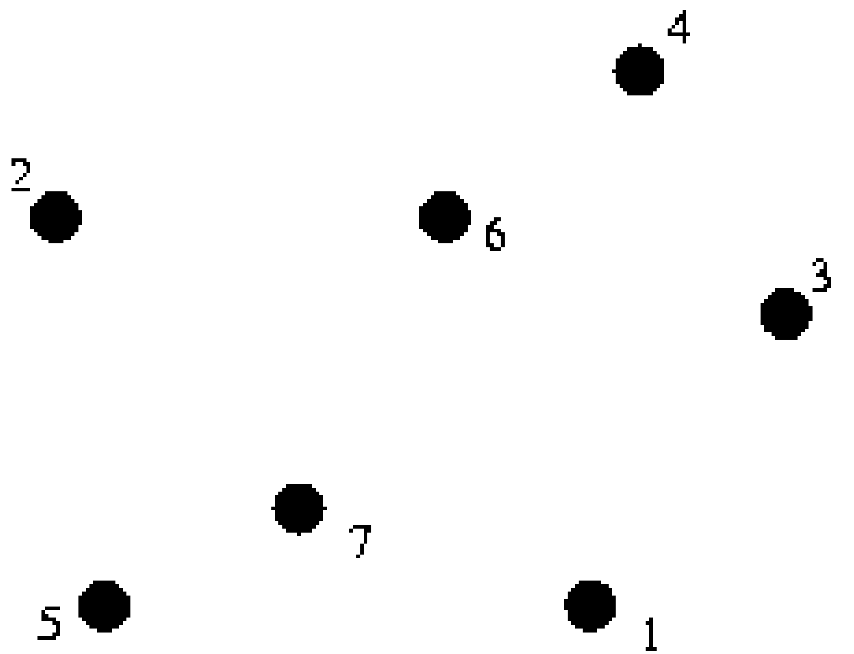
Circuit: 1



Stack:

Location:

Circuit: 1 5 7 6 4 3 7 2 6 5 2 4 1



Problem involving Eulerian Tour

- USACO Riding Fences
- Farmer John owns a large number of fences that must be repaired annually. He traverses the fences by riding a horse along each and every one of them (and nowhere else) and fixing the broken parts.
- Farmer John is as lazy as the next farmer and hates to ride the same fence twice. Your program must read in a description of a network of fences and tell Farmer John a path to traverse each fence length exactly once, if possible. Farmer J can, if he wishes, start and finish at any fence intersection.
- Every fence connects two fence intersections, which are numbered inclusively from 1 through 500 (though some farms have far fewer than 500 intersections). Any number of fences (>=1) can meet at a fence intersection. It is always possible to ride from any fence to any other fence (i.e., all fences are "connected").
- Your program must output the path of intersections that, if interpreted as a base 500 number, would have the smallest magnitude.
- There will always be at least one solution for each set of input data supplied to your program for testing.