DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING
CONCORDIA UNIVERSITY
COMP 6651 Algorithm Design Techniques
Winter 2024
Project Description
Due: Tuesday, April 16, 2024

Contents

1 Project Specification

In class, we studied the problem of finding a shortest path between any two nodes in a graph for which there are polynomial time algorithms as long as there are no negative weight cycles. For this project, instead, we will study the problem of finding a longest simple path in a given arbitrary undirected graph.

Consider the problem of finding a longest simple path on a given arbitrary undirected unweighted graph $G = (V, E)$. A path is called simple if it does not have any repeated vertices; the length of a path is measured by its number of edges in an unweighted graph. Denote P the set of simple paths in G. We are considering unweighted graphs, so the length (or cost) of a path $p \in P$ is defined to be the number of edges e of p: $w(p) = \sum_{e \in p} 1$. A longest simple path (LSP) problem consists of finding a path $p \in P$ such that $w(p)$ is maximal.

It turns out that the LSP problem is a computational difficult problem. The LSP problem can actually be shown to be NP-hard because the problem of deciding whether or not there exists a Hamilton tour in a given graph (an NP-complete problem) can be reduced to the LSP problem [1].

In this project, as a team, you will do the following:

1. You will implement a random graph generator to create an undirected unweighted geometric graph having certain properties, which will be stored in an external file using the ASCII-based EDGES format.
2. This project will use three randomly generated graphs using your random Euclidean graph generator, and three graphs taken from an online network repository.
3. You will determine the largest connected component (LCC) of each graph.
4. You will then run four heuristic LSP algorithms (one of your design) on the LCC of each graph, generating several metrics of your results.

5. These metrics will be entered into tables, as shown below.

You may use a programming language of your choice, such as Python, Java, or C++, to create a functional implementation of the required algorithms.

1.1 Required implementations:

- Random Euclidean Graph Generator

  For this project, you will need to randomly generate Euclidean neighbor graphs for the simulations to explore the behaviour of your different heuristic LSP algorithms.

  Consider the following pseudocode to generate a graph of n nodes with maximum distance r between node neighbors. After creating the n nodes, each node is assigned random (x, y) coordinates between 0 and 1. For each pair of vertices u and v within distance r, edges (u, v) and (v, u) are added to E. This creates an Euclidean neighbor graph.

```
GENERATE_GEOMETRIC_GRAPH(n, r)
  // n is number of vertices, r is maximum distance between nodes sharing an edge
  Define a set of vertices V such that |V| = n
  // assign each node in V random Cartesian coordinates (x, y) as follows
  for each vertex u ∈ V do
      u.x ← a uniform random number in [0..1]
      u.y ← a uniform random number in [0..1]
  // add all undirected edges of length ≤ r to E
  for each vertex u ∈ V do
      for each vertex v ∈ V do
          if (u ≠ v) && ((u.x − v.x)^2 + (u.y − v.y)^2 ≤ r^2) then
              E ← E ∪ {(u, v)}
```

  Generate one graph for each set of n and r values (detailed instructions below on how to choose n and r) and store the graph in an EDGES format file. This is an ASCII readable edge list format storing a graph as node pairs with no data, e.g., the undirected edge $(1, 2)$ is stored on one line of the file as:

```
1 2
```

  Note that this format cannot directly store isolated nodes with no neighbors. When running the simulations below, read in the stored Euclidean graph from the file to then use it.

- Heuristic LSP Algorithms

  Below are three algorithms that determine long simple paths in graphs that will serve the bases for heuristic LSP algorithms for your simulation results. The second and third of these algorithms are variations of shortest path algorithms.

Heuristic 1: DFS

The first approach to find an LSP in a graph will use depth-first search (DFS) (§20.3 of CLRS).

```
DFS(G)
  // G: Graph
  for each vertex u ∈ G.V do
      u.color ← WHITE
      u.π ← NIL
  time ← 0
  for each vertex u ∈ G.V do
      if u.color = WHITE then
          DFS-VISIT(G, u)

DFS-VISIT(G, u)
  time ← time + 1                          ▷ white vertex u has just been discovered
  u.d ← time
  u.color ← GRAY
  for each vertex v ∈ G.Adj[u] do          ▷ explore each edge (u, v)
      if v.color = WHITE then
          v.π ← u
          DFS-VISIT(G, v)
  time ← time + 1
  u.f ← time
  u.color ← BLACK                          ▷ blacken u; it is finished
```

How to use DFS to define a heuristic algorithm to find a longest simple path is discussed on Page 7.

Heuristic 2: Maximal path length Dijkstra

Consider the following modification[1] of Dijkstra's algorithm (§22.3 of CLRS) to try to find long simple paths in an undirected unweighted graph, starting at the source node s. Nodes with the largest $u.d$ values are expected to be terminal nodes of long simple paths.

```
Initialize-Single-Source-MAX(G, s)
  for each vertex v ∈ G.V do
      v.d ← −∞
      v.π ← NIL
  s.d ← 0

Relax-MAX(u, v)
  if v.d < u.d + 1 then                          ▷ w(u, v) = 1 (weight of edge (u, v))
      v.d ← u.d + 1
      v.π ← u

Dijkstra-MAX(G, s)
  // G: Graph; s: source node
  Initialize-Single-Source-MAX(G, s)
  S ← ∅
  Q ← ∅                                          ▷ A max heap.
  for each vertex u ∈ G.V do
      Insert(Q, u)
  while Q ≠ ∅ do
      u ← Extract-Max(Q)
      S ← S ∪ {u}
      for each vertex v ∈ G.Adj[u] do
          Relax(u, v)
          if the call of Relax-MAX increased v.d then
              Increase-Key(Q, v, v.d)
```

How to use this variation of Dijkstra to define a heuristic algorithm to find a longest simple path is discussed on Page 7.

[1] I give no guarantee that this modification is correct; if there are errors, you should make any necessary corrections.

Heuristic 3: Maximal path length A*

Consider the following modification[2] of the A* algorithm to try to find a longest simple path in an undirected unweighted graph, starting at the source node s and ending at a destination node d (which means that the destination node must be known, unlike with Dijkstra).

A* is a widely used pathfinding algorithm used in robotics, video games, etc., where reasonably short paths must be found quickly. This is achieved by using a heuristic value that estimates the remaining distance from the current node to the destination node. With regular A* pathfinding, the heuristic guides the search of the shortest path toward the destination. In our modification, we will guide the search *away* from the destination node. In the pseudocode below, we will use the Euclidean heuristic.

```
A_Star(G, s, d)
  // G: Graph; s: source node; d: destination node
  Initialize-Single-Source-MAX(G, s)
  for each vertex v ∈ G.V do                          ▷ set the heuristic distance from v to d
      v.h ← √((d.x − v.x)² + (d.y − v.y)²)             ▷ this is the Euclidean heuristic
  S ← ∅                                                ▷ closed list
  Q ← ∅                                                ▷ A max heap.
  for each vertex u ∈ G.V do
      Insert(Q, u)          ▷ The key value for node u in the max heap is u.d + u.h
  while Q ≠ ∅ do
      u ← Extract-Max(Q)
      S ← S ∪ {u}           ▷ even if u is d, we don't stop since we might find a longer path to d
      for each vertex v ∈ G.Adj[u] do
          Relax(u, v)
          if the call of Relax-MAX increased v.d then
              if v ∈ S then
                  S ← S − {v}                          ▷ remove v from closed list
                  Insert(Q, v)                         ▷ insert v back into open list
              else
                  Increase-Key(Q, v, v.d + v.h)
```

Note that since a vertex can be reintroduced to the priority queue, it is possible that the above pseudocode may result in loops (which obviously means the path found is not simple). If so, consider a modification where can not occur (you may want to consider the predecessor of a node before adding it back to the priority queue). You may also want to check that your final long path is simple and does not contain any cycles.

How to use this variation of A* to define a heuristic algorithm to find a longest simple path is discussed on Page 7.

[2] I give no guarantee that this modification is correct; if there are errors, you should make any necessary corrections.

1.2 Simulations and Results

Define the following metrics to be computed (one set of values for each heuristic and graph combination, to be stored in a table; see below):

- n: number of nodes in graph G, or $n = |V|$
- $|V_{LCC}|$: number of nodes in the largest connected component, LCC, of G
- $\Delta(LCC)$: the maximum degree of any node in $\|LCC\|$, or

$$\Delta(LCC) = \max_{v \in LCC} k(v)$$

  where $k(v)$ is the degree of node v
- $\overline{k(LCC)}$: the average degree of nodes in $\|LCC\|$, or

$$\overline{k(LCC)} = \frac{1}{|V_{LCC}|} \sum_{v \in LCC} k(v) = \frac{2 \cdot |E_{LCC}|}{|V_{LCC}|}$$

  where $|E_{LCC}|$ is the number of (undirected) edges in the largest connected component, LCC, of G
- L_{max}: length of LSP found by heuristic in the LCC (in terms of number of edges along the path)

Simulations

For this project, you are to perform simulations using three graphs from an online repository and three graphs that you will generate using your random Euclidean graph generator.

Online repository graphs

The first three graphs are from an online repository [2]:

1. DSJC500-5: A connected graph with 500 nodes and 62,600 edges with maximum degree 286 and average degree of 250.
   https://networkrepository.com/DSJC500-5.php
2. Euroroad: A connected graph with 1200 nodes and 1400 edges with maximum degree of 10 and average degree of 2.
   https://networkrepository.com/inf-euroroad.php
3. inf-power: A connected graph with 5000 nodes and 6600 edges with maximum degree 19 and average degree of 2.
   https://networkrepository.com/inf-power.php

Download each graph from the online repository using the URL listed with the graph. These graphs are stored in the EDGES format file or the or Matrix Market (MTX) file format. Both of these formats are symmetric file formats (as used for these graphs) for undirected graphs. That means, when you add edge (i, j) to your graph, you also add (j, i), although the file only stores (i, j).

For these graphs, apply depth-first search (DFS) to find the largest connected component (LCC). You will have to make some additions/changes to the DFS pseudocode to be able to compute the LCC.

Randomly generated graphs

Using your implementation of **GENERATE_GEOMETRIC_GRAPH**, generate random graphs with the following characteristics:

4. $n = 300$ and $0.9 \cdot n \leq |V_{LCC}| \leq 0.95 \cdot n$
5. $n = 400$ and $0.8 \cdot n \leq |V_{LCC}| \leq 0.9 \cdot n$
6. $n = 500$ and $0.7 \cdot n \leq |V_{LCC}| \leq 0.8 \cdot n$

Experimenting with different choices of r (hint, try binary search), generate a random graph. Using the graph generated, $G(V, E)$, apply depth-first search (DFS) to find the largest connected component (LCC). When you find a random graph that match the characteristics listed for that graph, store it in an EDGES file.

Heuristic LSP search

One the six graphs lists above, you will run four heuristic LSP search algorithms to determine L_{max} for heuristic on each graph:

1. DFS heuristic

   You have to take the DFS pseudocode on Page 3 and use it as the basis of a heuristic LSP search algorithm. The following is a possible approach:

   DFS-based_longest_simple_path(G)
   // G: Graph
   Determine LCC from G
   $L_{max} \leftarrow 0$
   for $i \leftarrow 1, \sqrt{|V_{LCC}|}$ **do**
   Randomly select a vertex u of V_{LCC}
   Use **DFS**(u) to find the vertex v at greatest depth $v.depth$ in DFS tree
   Use **DFS**(v) to find the vertex w at greatest depth $w.depth$ in DFS tree
   $L_{max} = \max\{L_{max}, v, depth, w.depth\}$

2. Dijkstra heuristic

   You have to take the modified Dijkstra pseudocode on Page 4 and use it as the basis of a heuristic LSP search algorithm.

3. A* heuristic

   You have to take the modified A* pseudocode on Page 5 and use it as the basis of a heuristic LSP search algorithm. This algorithm will only work for your geometric graphs, not the repository graphs. Only give experimental results for geometric graphs.

4. Your heuristic

   Find your own shortest path or pathfinding algorithm, modify it into an algorithm to find an LSP in a graph. For example, variants of A* are listed on the A* search algorithm Wikipedia page at https://en.wikipedia.org/wiki/A*_search_algorithm, or you could look at Amit's A* blog at http://theory.stanford.edu/~amitp/GameProgramming/. Use your modified algorithm as the basis of a heuristic LSP search algorithm.

   If your algorithm requires the use of a geometric graph (i.e., it uses the position information of the nodes in the graph), then only give experimental results for geometric graphs.

Results

Record your results in a table such as the following (such as might be presented for one of the graphs that you randomly generated (no, these values of n and r are not valid, so don't copy them).

Algorithm	n	r	$\lvert V_{LCC}\rvert$	$\Delta(LCC)$	$\overline{k(LCC)}$	L_{max}
DFS heuristic	300	0.1				
Dijkstra heuristic	300	0.1				
A* heuristic	300	0.1				
Your heuristic	300	0.1				

Table 1: Example results table for random graph with $n = 300$.

Below is an example results table for one of the online graphs, in this case, the Euroroad graph. Don't assume that the LCC is the entire graph for the online graphs. Only include your algorithm in this table if it does not require the use of a geometric graph (i.e., it only uses the node connection information and does not use position information of nodes in the graph).

Algorithm	n	r	$\lvert V_{LCC}\rvert$	$\Delta(LCC)$	$\overline{k(LCC)}$	L_{max}
DFS heuristic	1200	–				
Dijkstra heuristic	1200	–				
Your heuristic	1200	–				

Table 2: Example results table for Euroroad graph.

2 Project Deliverables

1. A project report, submitted in PDF format. This report should be concise and clearly written and include the following sections:

   - Problem description
     In your words, what is the purpose of this project?
   - Implementation details
     Give details of any of the main function implementations **including pseudocode**. Repeating back the instruction sheet will only get part marks.
   - Implementation correctness
     How your tested the correctness of your implementations, both in terms of software correctness (e.g., boundary testing, etc.) but also that your graph algorithms are correct (e.g., demonstrating correctness on small example graphs where the output can be verified independently of the program).
   - Results
     Complete the tables as requested and including explanation of any choices you made during your simulations and how they affected the results.
   - Conclusions
     What overall conclusions can you draw from the results of your simulations?

- Team work distribution
  In this part, clearly write down how the work was distributed to each member for evaluation purposes.
- References
  Any references used to implement the algorithms in your project. You must not "cut'n'paste" code from any of sources.

2. A zip file containing:
   - ASCII encodings of all your source-sink graphs as input to your simulation code, one file per graph.
   - Your implementation code including a README to explain how to compile and run.

3 Marking Details

It should be clear that the results of your simulations were derived from running your implementation code. The marker needs to be able to verify this. If this is not easily done, the following penalties may be applied:

- A deduction of at least 50% if your code does not run
- A deduction of at least 20% if it is very difficult to verify that the results of your simulations presented in your report were derived from running your code

4 Other Specifications

- Team size: ≤ 3. You are responsible to form your own team. On the first page of your report, *clearly* list the members of your team along with their student numbers.
- You may not use external libraries such as the Python libraries NetworkX and igraph for any part of your project. You must implement all aspects of your solution and can only use the built-in libraries that come with your chosen language.

References

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition ed., 1979.

[2] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.