5.3 Finding the shortest loop

Finding shortest cycle in graphs is a fundamental problem, but less explored compared to finding the shortest path. Efficient algorithm to find shortest cycle of a graph, also known as girth[89], is critical in cycle theory, determination of minimal cycle basis[90–92] and maximum cycle packing[93–95]. In particular, efficiently finding girth is crucial for methods that iteratively populate all loops with selected weights[?] . The shortest cycle problem is also related to graph properties such as chromatic number and connectivity[96;97], also for planar graphs it corresponds to the min-cut problem in the dual graph.

We can use all pair shortest path algorithm (Floyd-Warshall) to find the shortest cycle in directed graphs. However, this procedure is not directly applicable on undirected graphs due to possible self-loops and the objective of finding simple cycles without repetition of edges. For unweighted graphs Itai *et al.* used reductions in subcubic time (upperbounded by the matrix multiplication exponent); they left the weighted graphs[98] as an open problem. Roditty *et al.* extended Itai *et al.* result for integer weights[99].

Vassilevska *et al.* relates finding minimum cycle to other graph theory problems with no subcubic time algorithm[100] and thus any improvements for one of these problems influences others.

Further approximation algorithms are proposed in[98;101–104]. Finding shortest cycle with even length is analyzed in[105], and randomized algorithms proposed, e.g., in[103]. In this section, we focus on a deterministic algorithm to find the girth that has arbitrary length.

Because more efficient combinatorial algorithm is required for (real valued) weighted and undirected graphs, this research introduces an easily implementable method that incorporates the known shortest path algorithm philosophy. We employ a unique definition for walks with sockets (pair consisting of a vertex and an edge) and modify the existing Dijkstra's algorithm respectively. We translate the algorithm into nodes and edges afterwards.

Most of the proposed algorithms are focused on finding the shortest cycle rooted to each node and repeating the process for all nodes, e.g. in[98;103;106]. Nevertheless, we focused on minimizing a *composite distance* from each node to the cycles, with shrinking the network

Figure 5.1: Socket $\mathcal{S}_{a,e}$ consisting of vertex a and edge e.

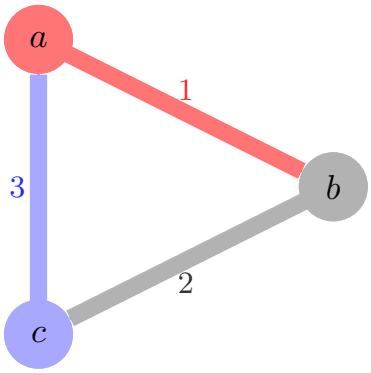

Figure 5.2: Chain of sockets $\mathcal{S}_{a,1}\mathcal{S}_{b,2}\mathcal{S}_{c,3}$ form a loop on vertices a, b, and c.

with discarding nodes that are not improving the subsequent searches.

Let $G = (V, E)$ be a graph with vertex set V and edge set E. The size of V and E are denoted by n and m subsequently. The set of neighbors of node v is denoted by $N_v = \{u \in V | u \text{ is connected to } v\}$. A *walk* γ on a graph is represented by a finite string of vertices and edges $v_1e_1v_2e_2 \dots v_r$, where v_i's and e_i's are all distinct. A *cycle* is a walk that starts and ends at the same vertex $v_1e_1v_2e_2 \dots v_re_rv_1$. The length of cycle c is defined as

$$\ell(c) := \sum_{e \in c} w(e), \tag{5.5}$$

where $w : E \to \mathbb{R}^+$ is a weight function, interpreted as length of edges.

A *socket* $\mathcal{S}_{v_i,e_j}$ includes a connected pair of vertex and edge (v_i, e_j), as shown in Figure 5.1. *Distinct sockets*, have no nodes or edges in common and weight of a socket $\mathcal{S}_{v_i,e_j}$ is weight of e_j.

Therefore, walk can be redefined with sockets; a finite string of distinct sockets that do not share vertices and edges, i.e., $\mathcal{S}_{v_1,e_1}\mathcal{S}_{v_2,e_2} \dots \mathcal{S}_{v_r,e_r}$ is a simple walk, and a cycle is a simple walk that starts from an initial socket $\mathcal{S}_{v_1,e_1}$ and ends in $\mathcal{S}_{v_r,e_r}$, where e_1 and e_r are two distinct edges incident to vertex v_1. A simple loop with sockets is shown in Figure 5.2.

5.3.1 Composite distance

Let $\mathcal{L}$ be the set of simple cycles in G; assume $\mathcal{L}$ is nonempty, i.e., G is not a tree. For $x, y \in V$ and $c \in \mathcal{L}$, let $d(x, y)$ be the distance between x and y, and $d(x, c)$ be the distance between x and c, that is

$$d(x, c) = \min_{y \in c} d(x, y).$$

Additionally, define the composite distance of node $x \in V$ to loop $c \in \mathcal{L}$:

$$d^+(x, c) = d(x, c) + \ell(c), \tag{5.6}$$

and the composite distance of node x to all loops $\mathcal{L}$:

$$d^+(x) = \min_{c \in \mathcal{L}} d^+(x, c). \tag{5.7}$$

The following theorem shows that we obtain the shortest cycle by solving the optimization problem (5.7) for every vertex x,

Theorem 5.3.1. *Minimizing $d^+(x)$ over $x \in V$ is equivalent to finding the shortest cycle in $\mathcal{L}$, and the minimum is attained for any x in a shortest cycle.*

Proof. For any $x \in V$ and $c \in \mathcal{L}$, $d^+(x, c) \geq \ell(c)$, and equality holds if $x \in c$, so the minimum will be attained when c is a shortest cycle and $x \in c$. □

Theorem 5.3.1 suggests to find a shortest cycle, we determine $d^+(x)$ for all nodes subsequently, with using the previous best $d^+(x)$ as a cut-off for the next. The following theorem shows that we can exclude node z that $d(x, z) \leq d(x, c)$ from further consideration in our search.

Theorem 5.3.2. *Suppose $d^+(x) = d^+(x, c)$ for some $x \in V$ and $c \in \mathcal{L}$. Let $c' \neq c \in \mathcal{L}$ be a shortest cycle and let $y \in c'$. Then*

$$d(x, y) > d(x, c).$$

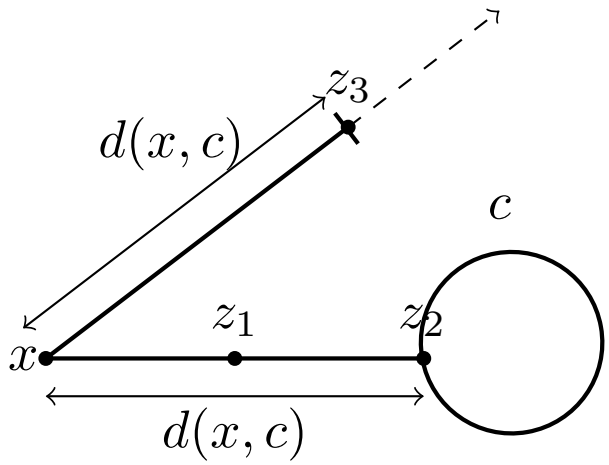

Figure 5.3: Nearest cycle to node x. Nodes such as z_1, z_2, and z_3 that $d(x, z_i) \leq d(x, c)$ can be excluded from further considerations.

In other words, after a search starting from x, we can exclude from further consideration any node that x is closer to them than the minimizing cycle c.

Proof. Since $d^+(x) < d^+(x, c')$ and $\ell(c') < \ell(c)$, we have

$$d(x, c) + \ell(c) < d(x, c') + \ell(c') < d(x, c') + \ell(c),$$

so

$$d(x, c) < d(x, c') < d(x, y),$$

implying the theorem. □

As a trivial example for Theorem 5.3.2, after a search from node v, all nodes z that $d(x, z) \leq d(x, c)$ will be excluded from subsequent searches (Figure 5.3).

5.3.2 Algorithm

The proposed method searches the sockets in the graph with Dijkstra's algorithm while using a priority queue implementation to map each socket to its position in the queue[107]. We illustrate an example of a cycle including vertex a with degree 3 that transforms into a cycle in Figure 5.4(a) to a path of distinct sockets from set $\{\mathcal{S}_{a,1}, \mathcal{S}_{a,2}, \mathcal{S}_{a,3}\}$ to $\{\mathcal{S}_{N_1,1}, \mathcal{S}_{N_2,2}, \mathcal{S}_{N_3,3}\}$

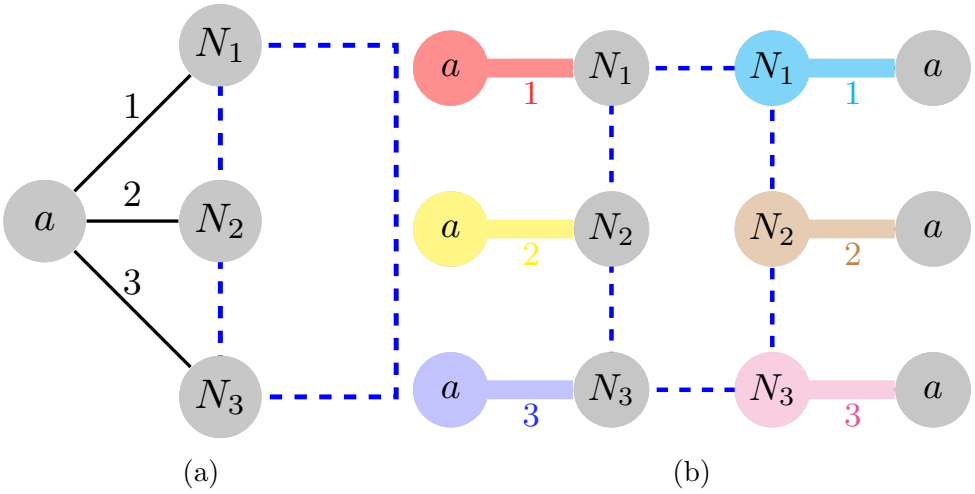

Figure 5.4: (a) Vertex a with neighbors. The remainder of the graph is shows by a dashed rectangle; (b) shortest path between set of sockets $\{\mathcal{S}_{a,1}, \mathcal{S}_{a,2}, \mathcal{S}_{a,3}\}$ to another set of sockets $\{\mathcal{S}_{N_1,1}, \mathcal{S}_{N_2,2}, \mathcal{S}_{N_3,3}\}$, such that all sockets are distinct, gives the shortest cycle for vertex a.

in Figure 5.4(b).

Following Theorem 5.3.1, to find a shortest cycle we examine the smallest found $d^+(x)$ over $x \in V$ by running Dijkstra's algorithm (see Algorithm 4). Whence the shortest path starting from sockets attached to x meets a nondistinct socket, algorithm returns $d^+(x)$.

We illustrate an example of a weighted graph in Figure 5.5(a) and the found $d^+(a)$ in Figure 5.5(b) with nodes that can be excluded from further consideration in yellow color. We demonstrate different steps of the algorithm in Figure 5.6.

To find the girth we apply Algorithm 4 for each node as root and use the shortest found cycle as a cut-off for the next search with excluding the nodes that cannot be roots for future searches using Theorem 5.3.2. The pseudo-code is shown in Algorithm 5, from the algorithms and theorems we conclude the following corollary:

Corollary 5.3.3. *Using Algorithm 4 and 5, each loop will be searched at most only once. Moreover, the number of loops that the algorithm completes is upperbounded by size of cycle basis*[108].

We translate the proposed algorithms into vertices and edges. Following Theorem 5.3.1, to find the girth we examine the smallest found $d^+(x)$ over $x \in V$ exhaustively using Dijkstra's algorithm: when a node z is added to the set of "visited" nodes (i.e., the nodes whose a

Algorithm 4 Algorithm to determine improved $d^+(x) = \min_{c \in C} d^+(x, c)$ and excluded sockets.

Q.**enqueue**(Adjacent sockets to x with their weights)
walk$\leftarrow$ Dictionary of shortest walk for each socket from source sockets
while Q **do**
    $\mathcal{S}$, dist($\mathcal{S}$) $\leftarrow$ Q.**dequeue**
    if dist($\mathcal{S}$) $>$ Cut-off **then**
        return None
    end if
    $u \leftarrow$ adjacent vertex to $\mathcal{S}$.
    if $u \in$ socket $\mathcal{T}$ in walk[$\mathcal{S}$] **then**
        return dist($\mathcal{S}$) and nodes in sockets with distance less than dist($\mathcal{T}$).
    end if
    for Sockets $\mathcal{R}_{u,e_i}$ starting from u and distinct from $\mathcal{S}$ **do**
        dist($\mathcal{R}_{u,e_i}$) = dist($\mathcal{S}$) + weight of $w(e_i)$)
        if dist($\mathcal{R}_{u,e_i}$) $>$ Cut-off **then**
            continue to next iteration
        end if
        if $\mathcal{R}_{u,e_i} \notin$ seen or dist($\mathcal{R}_{u,e_i}$) $<$seen[$\mathcal{R}_{u,e_i}$] **then**
            seen[$\mathcal{R}_{u,e_i}$] $\leftarrow$ dist($\mathcal{R}_{u,e_i}$)
            update walk[$\mathcal{R}_{u,e_i}$] = walk[$\mathcal{S}$] + $\mathcal{R}_{u,e_i}$
            Q.**enqueue**($\mathcal{R}_{u,e_i}$, dist($\mathcal{R}_{u,e_i}$))
        end if
    end for
end while
return None

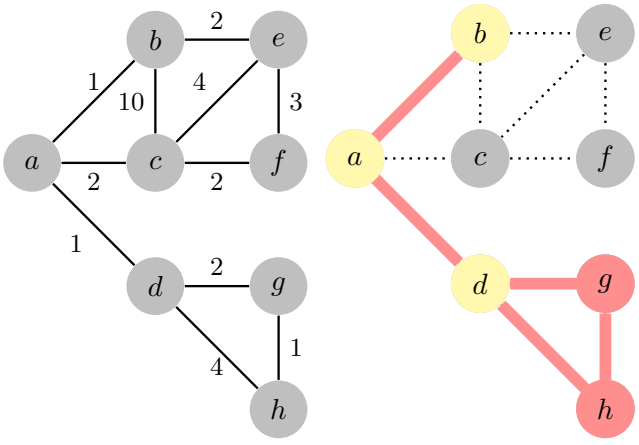

Figure 5.5: (a) Example graph with weights. (b) Nearest loop to vertex a, i.e., $\arg\min_{c \in C} d^+(x, c)$. Sockets linked to nodes a, b, and d will be discarded for the subsequent searches.

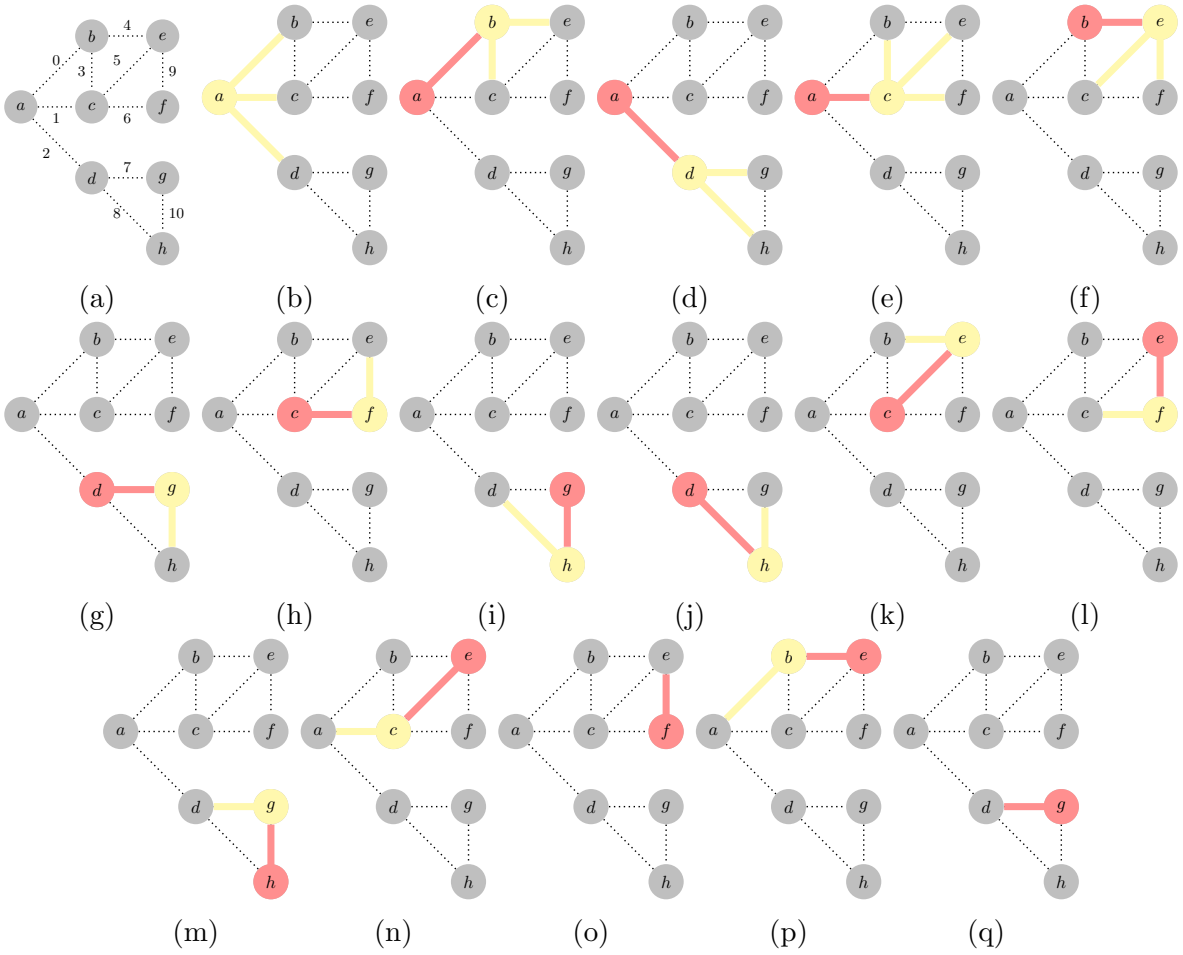

Figure 5.6: Solution steps in which each yellow socket is pushing into the queue and red sockets are popping. (a) Enumerating edges for better explanation (unnecessary for the proposed algorithm); (b) pushing the immediate attaching socket to vertex a with their weights as priority: $(\mathcal{S}_{a,0}, 1)$, $(\mathcal{S}_{a,2}, 1)$, and $(\mathcal{S}_{a,1}, 2)$; (c) $(\mathcal{S}_{a,0}, 1)$ popped and neighboring sockets with new distances pushed: $(\mathcal{S}_{b,3}, 11)$ and $(\mathcal{S}_{b,4}, 3)$; (d) $(\mathcal{S}_{a,2}, 1)$ popped, and $(\mathcal{S}_{d,7}, 3)$, and $(\mathcal{S}_{d,8}, 5)$ pushed; (e) $(\mathcal{S}_{a,1}, 2)$ popped and $(\mathcal{S}_{c,3}, 12)$, $(\mathcal{S}_{c,5}, 6)$, and $(c_{c,6}, 4)$ pushed. (f) $(\mathcal{S}_{b,4}, 3)$ popped and $(\mathcal{S}_{e,5}, 7)$, and $(\mathcal{S}_{e,9}, 6)$ pushed; (g) $(\mathcal{S}_{d,7}, 3)$ popped and $(\mathcal{S}_{g,10}, 4)$ pushed; (h) $(c_{c,6}, 4)$ popped and $(\mathcal{S}_{f,9}, 7)$ pushed; (i) $(\mathcal{S}_{g,10}, 4)$ popped and $(\mathcal{S}_{h,8}, 8)$ pushed; (j) $(\mathcal{S}_{d,8}, 5)$ popped and $(\mathcal{S}_{h,10}, 6)$ pushed; (k) $(\mathcal{S}_{c,5}, 6)$ popped and $(\mathcal{S}_{e,4}, 8)$ pushed; (l) $(\mathcal{S}_{e,9}, 6)$ popped and $(\mathcal{S}_{f,6}, 8)$ pushed; (m) $(\mathcal{S}_{h,10}, 6)$ popped and $(\mathcal{S}_{g,7}, 8)$ pushed; (n) $(\mathcal{S}_{e,5}, 7)$ popped and $(\mathcal{S}_{c,1}, 9)$ pushed; (o) $(\mathcal{S}_{f,9}, 7)$ popped but does not lead to any socket from a previously shorter walk; (p) $(\mathcal{S}_{e,4}, 8)$ popped and $(\mathcal{S}_{b,0}, 9)$ pushed; (q) $(\mathcal{S}_{f,6}, 8)$ popped and interrupts its shortest path. The algorithm returns the closest circle: $\mathcal{S}_{d,8}, \mathcal{S}_{h,10}, \mathcal{S}_{g,7}$ with length $4 + 1 + 2 = 7$, shown in Figure 5.5(b).

Algorithm 5 Using Algorithm 4 to find the girth

```
cut-off ← ∞
while V do
    v ← V.pop
    c, Y ← Algorithm 4 for v, and with cut-off
    remove Y from G
    if cut-off > ℓ(c) then
        cut-off = ℓ(c)
    end if
end while
return cut-off
```

path from x to them is now determined). For each neighbor y of z, if a shorter path is found from the root, we update the shortest walk to y. If y has been visited and the found shortest path to y cannot be improved, we have a candidate for $d^+(x)$ of the root node x, namely the walk that begins at x, follows the Dijkstra tree to z crosses $\{z, y\}$ and returns to x along the Dijkstra tree. The candidate composite distance is $d(x, z) + d(x, y) - d(x, ca) + w(\{x, y\})$, where ca is the common ancestor of z and y in the shortest path tree. Algorithm enqueues the found candidate for the composite distance to the priority queue. Whenever a candidate dequeues by the algorithm, a composite distance $d^+(x)$ is found, finishing the search. As we run Dijkstra, we keep track of the best $d^+(x)$ found so far as a cut-off for the next search. If, at any point, we visit a node z that is farther from x than this best distance, we stop the search and excludes nodes that are not improving further searches by Theorem 5.3.2.

In the following theorem, we obtain the algorithm complexity.

Theorem 5.3.4 (Complexity of the algorithm)**.** *The worst case complexity for finding the girth from*

$$\min_{x \in V} \left(\min_{c \in C} d^+(x, c) \right) \tag{5.8}$$

with algorithms 4 and 5 using socket is $\mathcal{O}(\langle k \rangle n^2 \log n)$, where $\langle k \rangle$ is the average degree of the network and with nodes and edges is in $\mathcal{O}(nm + n^2 \log n)$ or.

Proof. In the socket language; from a node v, algorithm starts searching from all adjacent sockets and at most completes a tree with an extra socket to close a loop, i.e., $n - 1$ sockets

plus one. In each step i, from Theorem 5.3.2, algorithm excludes nodes that cannot improve the found shortest cycle and thus the network is shrinking. Therefore, the total number of heap operations is

$$\sum_{i=1}^{n} k^*(i)(n-i+1) \tag{5.9}$$

where $k^*(i) : \{1 : N\} \to \mathbb{Z}$ is the degree at step i. Equation (5.9) is bounded by $nk_{\max}$ because

$$\sum_{i=1}^{n} k^*(i)(n-i+1) < n\sum_{i=1}^{n} k^*(i) = nm$$

Therefore the worst case complexity, with enqueuing heap operation in $\mathcal{O}(\log(n))$, is in $\mathcal{O}(\langle k\rangle n^2 \log n)$. With nodes and edges it is in $\mathcal{O}(nm + n^2 \log n)$ □

5.3.3 Examples

We consider test examples to compare the algorithm with the naive counterparts where to find the girth in the graph is for all edges $e = \{u, v\} \in E$, we find the shortest path γ from u and v, such that $e \notin \gamma$, the resultant cycle $\gamma + e$, is the shortest cycle rooted to e. Repeating this process for all edges and comparing the length of them results in choosing the shortest one in $\mathcal{O}(m^2 + nm \log n)$ using Dijkstra's algorithm with Fibonacci heap operations.

In the first example, we consider a random geometric graph to illustrate the algorithm behavior. Moreover, because complete graphs (or similar graphs) are posing a challenge for the algorithm performance due to their large degrees, we test them as the second example.

Random geometric graph with light spanning tree An example for short-circuiting in networks can be described in a random geometric network G, with weight values chosen uniformly random between 10^4 and 10^5. After finding a spanning tree $T = (V, E_T)$, we reweigh the edges E_T to be uniformly random between 20 and 50. Now if we randomly choose one edge in $E \setminus E_T$ and assign a small weight, the shortest cycle comprises this edge and some edges in E_T. Finding this cycle is hard for the naive algorithm because it finds the shortest cycle rooted to each edge. However, our proposed algorithm finds this shortest