Finding a Graph with Minimum Cut Less Than Its Minimum Degree

Matt Holiday

21 May 2019

Abstract

What is the nature of a graph where the minumum cut is less than the minimum degree of any node? This program finds example random graphs with that property.

1 Purpose

The minimum cut of a graph is the number of links (or edges) in the graph which must be removed to disconnect (or partition) the graph into two subgraphs. A graph with node of degree 0 or a graph that is initially disconnected will have a minimum cut of 0. Also, the upper bound for the minimum cut of a graph is the minimum degree among all nodes in the graph, as removing all links from that node causes it to be disconnected.

The question is, what type of graph will have a minimum cut greater than 0 but less than the minimum degree? The following program is intended to find example graphs meeting that criteria.

1.1 The minimum-cut algorithm of Stoer and Wagner [and Frank]

The algorithm proceeds by picking a key node and "contracting" into it the adjacent node with the greatest number of links into that key node. Although the graph is not a multigraph at the beginning (we could allow this, but didn't), it will become one as as nodes are contracted. Links to the node being contracted away are relabeled to link to the key node, retaining duplicate links while eliminating self-links (see the `node` struct). At each iteration, we track the degree of the key node whose links can only be to the remainder of the graph; the minimum value over all steps is the minimum cut in the

graph overall. This algorithm is simpler than the one in the original paper[1] because we sort the node struct's list of linked nodes rather than use a priority queue.

```
func findMaxAdjacent(key node, nodes []node) ([]node, node) {
    b := &byAdjacency{nodes: nodes, key: key}
    l := len(b.nodes) - 1

    sort.Sort(b)
    return b.nodes[:l], b.nodes[l]
}

func contract(key *node, last node, nodes []node) []node {
    key.contract(last)

    for _, n := range nodes {
        n.swap(last, *key)
    }

    key.swap(last, *key)
    return nodes
}

func findMinCut(nodes []node, min, goal int, w io.Writer) (int, []cut) {
    cuts := make([]node, 0, len(nodes))
    var key, last node
    var all []node

    key, all = nodes[0], nodes[1:]

    for len(all) > 1 {
        fmt.Fprintln(w, "key=", key, "nodes=", all)

        all, last = findMaxAdjacent(key, all)
        all = contract(&key, last, all)
        cuts = append(cuts, last)

        if deg := key.degree(); deg < min {
            min = deg
        }

        fmt.Fprintln(w, "key=", key, "last=", last, "nodes=", all)

        if min <= goal {
            return min, reduce(key, cuts, w)
        }
    }

    return min, nil
}
```

[1] M. Stoer and F. Wagner, "A simple min cut algorithm," *Journal of the ACM,* 44 (1997): 585–591; see also A. Frank, "On the edge-connectivity algorithm of Nagamochi and Ibaraki," Laboratoire Artemis, IMAG, Université J. Fourier, Grenoble, 1994.

While `findMinCut` calculates the minimum cut, it does not by itself identify the edges in the original graph which make up that cut. We know what the key node connects to in the final contracted graph, but the contractions may have renumbered edges. Thus we take the final key node's links and the list of contractions so we can renumber the links from the key node and make a cut list of edges.

```
type cut struct {
    l, r int
}

func reduce(key node, cuts []node, w io.Writer) []cut {
    fmt.Fprintln(w, "cuts=", cuts)

    find := func(v int, in []int) (int, bool) {
        for i, j := range in {
            if v == j {
                return i, true
            }
        }
        return -1, false
    }

    final := []cut{}

    // when we find the size of the minimum cut, the links from the key node
    // may have been contracted, so we're going to work backward through the
    // contractions to identify the actual nodes that were contracted into the
    // key node so we can relabel the links; we try to find the last removed
    // node with a link to a remaining node from the key node, and substitute

    for _, l := range key.links {
        found := false

        for i := len(cuts) - 1; i >= 0; i-- {
            // see if the cut's RHS appears contains the link

            if _, ok := find(l, cuts[i].links); ok {
                final = append(final, cut{cuts[i].id, l})
                found = true
                break
            }
        }

        // we didn't find a replacement, so it must have
        // been linked from the key node in the beginning

        if !found {
            final = append(final, cut{key.id, l})
        }
    }

    return final
}
```

1.2 Driver

The main program takes three arguments: the maximum size n of random graphs to generate, the maximum iterations for each graph size, and the minimum cut as a search goal. It will try graphs of size 6 up to n using link probabilities in the range 0.4–0.9. It will ignore disconnected graphs and graphs where the minimum cut is 0, i.e., graphs that are initially disconnected because a single node or a clique is separated from the rest of the graph. It will also ignore graphs where the minimum degree of any node is less than the goal since they have a trivial minimum cut: just remove the links to that node.

`searchMinCut` finds the degree of the graph and uses that as the upper bound for any minimum cut. Note that the method of constructing the graph (using a "triangle" of node pairs by their node ID) prevents the graph from having multiple links between two nodes, i.e., it will not be a multigraph at the beginning. Note that in order to print the original graph we make a deep copy because otherwise the nodes given to the algorithm will end up mutated by the contraction process.

```
func deepCopy(nodes []node) []node {
    all := make([]node, len(nodes))

    for i, n := range nodes {
        all[i].id = i

        for _, l := range n.links {
            all[i].links = append(all[i].links, l)
        }
    }

    return all
}

func searchMinCut(cnt, goal int, prob float32,
    w *bytes.Buffer) (minCt, minDg int, cuts []cut, nodes []node) {
    // build out the random graph, keeping track of the sum
    // of degrees which we use as a maximum possible cut

    nodes = make([]node, cnt)
    maxCt := 0

    for i := 0; i < cnt; i++ {
        nodes[i].id = i
    }

    for j := 0; j < cnt; j++ {
        for k := 0; k < j; k++ {
            maxCt += nodes[j].randLink(&nodes[k], prob)
        }
    }
```

```
    // find the minimum degree

    minDg = maxCt

    for i := 0; i < cnt; i++ {
        if deg := nodes[i].degree(); deg < minDg {
            minDg = deg
        }
    }

    // there's no point in searching a graph whose
    // min degree is too small

    if minDg <= goal {
        return 0, minDg, nil, nil
    }

    fmt.Fprintln(w, "\nNEW", "nodes=", nodes, "maxCt=", maxCt, "minDg=", minDg)

    // we must copy nodes so we can print it out later

    minCt, cuts = findMinCut(deepCopy(nodes), maxCt, goal, w)

    if minCt <= goal && minCt != len(cuts) {
        log.Fatal("invalid cut", cuts, w.String())
    }

    return minCt, minDg, cuts, nodes
}

func run(n, its, goal int) {
    probs := []float32{0.4, 0.5, 0.6, 0.7, 0.8, 0.9}

    for i := 6; i < n+1; i++ {
        for _, p := range probs {
            for j := 0; j < its; j++ {
                w := new(bytes.Buffer)
                minCt, minDg, cuts, nodes := searchMinCut(i, goal, p, w)

                if 0 < minCt && minCt < minDg {
                    fmt.Printf("minCt=%d, minDg=%d cuts=%v, nodes=%v\n",
                        minCt, minDg, cuts, nodes)
                    fmt.Println(w.String())
                }
            }
        }
    }
}
```

That leaves the main routine and the import list (assume all code is in one file):

```
package main
```

```
import (
    "bytes"
    "fmt"
    "io"
    "log"
    "math/rand"
    "os"
    "sort"
    "strconv"
    "time"
)

func main() {
    var n, its, goal int
    var err error

    if len(os.Args) < 4 {
        log.Fatal("not enough args")
    }

    if n, err = strconv.Atoi(os.Args[1]); err != nil {
        log.Fatal("invalid node count:", os.Args[1])
    }

    if its, err = strconv.Atoi(os.Args[2]); err != nil {
        log.Fatal("invalid iteration count:", os.Args[2])
    }

    if goal, err = strconv.Atoi(os.Args[3]); err != nil {
        log.Fatal("invalid min cut:", os.Args[3])
    }

    rand.Seed(time.Now().UnixNano())
    run(n, its, goal)
}
```

1.3 Nodes

The node struct represents a single node in the graph together with a list of IDs of linked nodes. We don't keep a list of node pointers because we'd have to do much copying to get around possible aliasing.

When the graph is created, we make a link from one node to another only if a random value in $[0, 1]$ is less than the link probability being tested. This ensures the graph is relatively sparse, but doesn't prevent a disconnected graph.

```
type node struct {
    id    int
    links []int
}
```

```
func (n *node) randLink(r *node, prob float32) int {
    if rand.Float32() < prob {
        n.links = append(n.links, r.id)
        r.links = append(r.links, n.id)

        return 1
    }

    return 0
}

func (n *node) degree() int {
    return len(n.links)
}

func (n *node) adjacency(r node) int {
    var rslt int

    // could be a multiple due to contraction

    for _, l := range n.links {
        if l == r.id {
            rslt++
        }
    }

    return rslt
}

func (n *node) swap(on, nn node) {
    for i, l := range n.links {
        if l == on.id {
            n.links[i] = nn.id
        }
    }
}

func (n *node) contract(r node) {
    n.links = append(n.links, r.links...)
    newlks := make([]int, 0, len(n.links))

    for _, l := range n.links {
        if l != n.id && l != r.id {
            newlks = append(newlks, l)
        }
    }

    n.links = newlks
}

type byAdjacency struct {
    nodes []node
    key   node
}
```

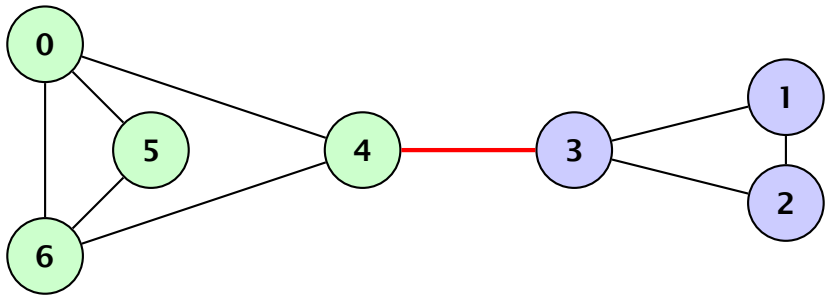
```go
func (b *byAdjacency) Len() int {
    return len(b.nodes)
}

func (b *byAdjacency) Less(i, j int) bool {
    return b.nodes[i].adjacency(b.key) < b.nodes[j].adjacency(b.key)
}

func (b *byAdjacency) Swap(i, j int) {
    b.nodes[j], b.nodes[i] = b.nodes[i], b.nodes[j]
}
```

2 Results

The algorithm found examples of size 7 with a minimum degree of 2 and minimum cut of 1. Matching graphs typically have a clique which is connected to the main body of the graph with fewer links than the minimum degree of any node. In the first example the clique of three nodes $[1, 2, 3]$ is connected by a single link to node 4 (see Figure 1).

```
minCt=1, minDg=2 cuts=[{4 3}],
nodes=[{0 [4 5 6]} {1 [2 3]} {2 [1 3]} {3 [1 2 4]} {4 [0 3 6]} {5 [0 6]} {6 [0 4 5]}]
```

Figure 1 Graph #1 with a clique of three nodes



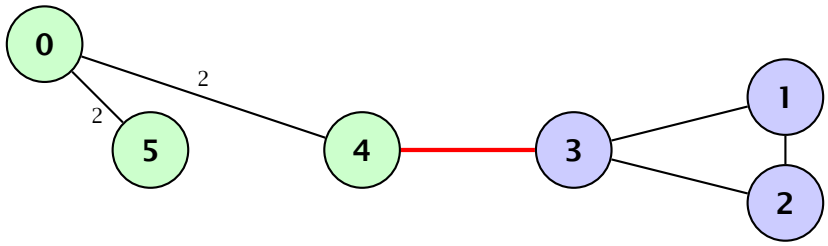
2.1 How we got the result for graph #1

We always start with node 0 being the key node, and contract other nodes one-by-one into it. In the first iteration, we select node 6 (one of the three nodes linked to node 0) to contract into node 0, which duplicates the links from node 0 to nodes 4 and 5 (see figure 2).

```
key= {0 [4 5 6]}
nodes= [{1 [2 3]} {2 [1 3]} {3 [1 2 4]} {4 [0 3 6]} {5 [0 6]} {6 [0 4 5]}]
```

```
key= {0 [4 5 4 5]}
last= {6 [0 4 5]}
nodes= [{1 [2 3]} {2 [1 3]} {3 [1 2 4]} {4 [0 3 0]} {5 [0 0]}]
```

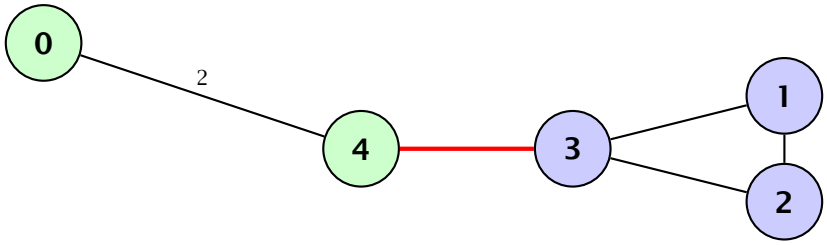
Figure 2 Graph #1 with node 6 removed



In the second iteration, we select node 5 due to its duplicate links to node 0. As a result, node 0 keeps its links to node 4 and drops node 5 (see figure 3).

```
key= {0 [4 5 4 5]}
nodes= [{1 [2 3]} {2 [1 3]} {3 [1 2 4]} {4 [0 3 0]} {5 [0 0]}]

key= {0 [4 4]}
last= {5 [0 0]}
nodes= [{1 [2 3]} {2 [1 3]} {3 [1 2 4]} {4 [0 3 0]}]
```

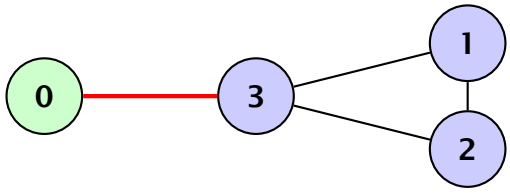
Figure 3 Graph #1 with nodes 5, 6 removed



In the third iteration, node 4 is contracted into node 0 leaving only the link to node 3. Thus, we've found the minimum cut (see figure 4).

```
key= {0 [4 4]}
nodes= [{1 [2 3]} {2 [1 3]} {3 [1 2 4]} {4 [0 3 0]}]

key= {0 [3]}
last= {4 [0 3 0]}
nodes= [{1 [2 3]} {2 [1 3]} {3 [1 2 0]}]
```

Figure 4 Graph #1 with nodes 4, 5, 6 removed



We then post-process the contractions that were made. We only have one link to relabel because the key node has only the one link; starting at the end of the list (the most recent contraction), we find a link to node 3 from node 4 in the last contraction (node 4 was contracted into the key node 0), so we renumber the 0–3 link as 4–3:

```
cuts= [{6 [0 4 5]} {5 [0 0]} {4 [0 3 0]}]
```

That leaves us our result, shown again with the single required cut between 4 and 3:

```
minCt=1, minDg=2 cuts=[{4 3}],
nodes=[{0 [4 5 6]} {1 [2 3]} {2 [1 3]} {3 [1 2 4]} {4 [0 3 6]} {5 [0 6]} {6 [0 4 5]}]
```