Advanced network algorithms, random graph models

You are given four networks in Pajek format (edge list and LNA formats are also available).

- A simple toy network for testing (tiny)
- The famous Zachary karate club network (small)
- iMDB actors collaboration network (medium)
- A part of Google web graph (large)

I. Number and size of connected components

1. Study the following algorithm for computing (weakly) connected components $\{C\}$ by any order link traversal. Does the algorithm implement breadth-first or depth-first search? What is the time complexity of the algorithm?

```
input   graph G, nodes N
output  network components {C}
  {C} ← empty list
  while not N empty do
      {C}.add(component(G, N, N.next()))
  return {C}
```

```
input   graph G, nodes N, node i
output  weak component C
  C ← empty list
  S ← empty stack
  N.remove(S.push(i))
  while not S empty do
      C.add(i ← S.pop())
      for neighbors j ∈ Γ_i do
          if N.remove(j) then
              S.push(j)
  return C
```

2. Try to implement the algorithm, and compute the number of (weakly) connected components s and the size of the largest (weakly) connected component S of all four networks. Are the results expected or are they surprising?

3. How could you further improve the algorithm to only compute s and S?

II. Average node distance and network diameter

1. Study the following algorithm for computing the distances between the nodes $\{d\}$ by level order link traversal. Does the algorithm implement breadth-first or depth-first search? What is the time complexity of the algorithm?

```
input   graph G
output  network distances {D}
  {D} ← empty list
  for nodes i ∈ N do
      {D}.add(distances(G, i))
  return {D}
```

```
input   graph G, node i
output  directed distances D
  ...
  for successors j ∈ Γ_i^out do
      ...
```

```
input   graph G, node i
output  undirected distances D
  D ← empty array
  Q ← empty queue
  D[Q.add(i)] ← 0
  while not Q empty do
      i ← Q.remove()
      for neighbors j ∈ Γ_i do
          if D[j] undefined then
              D[j] ← D[i] + 1
              Q.add(j)
  return D
```

2. Try to implement the algorithm, and compute the average distance between the nodes $\langle d \rangle$ and the maximum distance or diameter $d_{\max}$ for smaller networks. Are the results expected or are they surprising?

3. How is the algorithm different from the famous Dijkstra's algorithm? In which case you would have to use Dijkstra's algorithm?

4. How could you further improve the algorithm to only approximate $\langle d \rangle$ and $d_{\max}$?

III. Average node clustering coefficient

1. Study the following algorithm for computing the clustering coefficient of the nodes $\{C\}$ by link triad counting. Why does the algorithm count triads over the links? What is the time complexity of the algorithm?

```
input   graph G
output  average clustering ⟨C⟩
  ⟨C⟩ ← 0
  for nodes i ∈ N do
      ⟨C⟩ ← clustering(G, i)/n
  return ⟨C⟩
```

```
input   graph G, node i
output  node clustering C
  if k_i ≤ 1 then
      return 0
  return triads(G, i) · 2/(k_i^2 − k_i)
```

```
input   graph G, node i
output  node triads t
  t ← 0
  for neighbors j ∈ Γ_i do
      if |Γ_i| ≤ |Γ_j| then
          t ← t + triads(G, i, j)/2
      else
          t ← t + triads(G, j, i)/2
  return t
```

```
input   graph G, link i, j
output  link triads t
  t ← 0
  for neighbors k ∈ Γ_i do
      if k ∈ Γ_j then
          t ← t + 1
  return t
```

2. Try to implement the algorithm and compute the average node clustering coefficient $\langle C \rangle$ for all four networks. Are the results expected or are they surprising?

3. What kind of network representation is required by the algorithm?

IV. Erdös-Rényi random graphs and link indexing

1. Study the following two algorithms for generating Erdös-Rényi random graphs $G(n, m)$ with and without link indexing $\binom{i}{2} + j$, $i > j$. What is the main difference between the algorithms? What is the time complexity of the algorithms?

```
input   nodes n, links m
output  simple random G
    H ← empty set
    G ← n isolated nodes
    while not G has m links do
        h ← {0, ..., (n^2 − n)/2 − 1}.random()
        if H.add(h) then
            i ← 1 + ⌊−0.5 + √(0.25 + 2h)⌋
            add link between i and h − (i^2 − i)/2
    return G
```

```
input   nodes n, links m
output  random multi G
    G ← n isolated nodes
    while not G has m links do
        i, j ← {0, ..., n − 1}.random()
        if i ≠ j then
            add link between i and j
    return G
```

2. Try to implement one of the algorithms, and generate Erdös-Rényi random graphs corresponding to all four networks and compute their S, $\langle d \rangle$ and $\langle C \rangle$. Are the results expected or are they surprising?

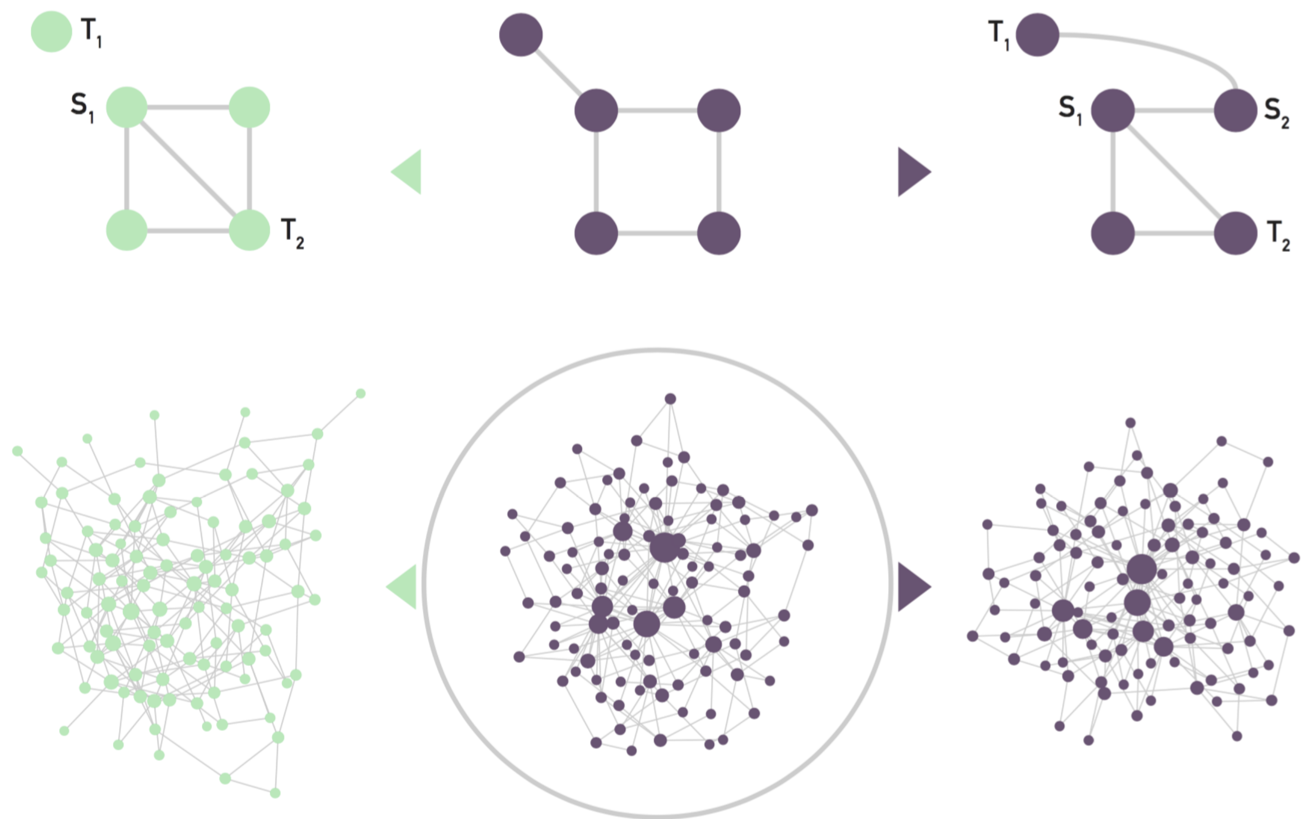
V. Configuration model graphs and link rewiring

1. Study the following two algorithms for generating configuration model graphs $G(\{k\})$ with link rewiring and stub matching. What is the main difference between the algorithms? What is the time complexity of the algorithms?

```
input   simple links L
output  configuration simple G
    H ← empty set
    for links {i, j} ∈ L do
        H.add(h_ij)
    while not links L rewired do
        {i, j}, {s, t} ← L.random()    ▷ removes links
        if H.contains(h_it or h_sj) or i = t or s = j then
            L.add({i, j})
            L.add({s, t})
        else
            L.add({i, t}) H.add(h_it) H.remove(h_ij)
            L.add({s, j}) H.add(h_sj) H.remove(h_st)
    return G on links L
```

```
input   nodes n, degrees {k}
output  configuration pseudo G
    Q ← empty queue
    G ← n isolated nodes
    for nodes i ∈ N do
        for k_i times do
            Q.add(i)
    while not Q empty do
        i, j ← Q.random()    ▷ removes nodes
        add link between i and j
    return G
```

2. Try to implement one of the algorithms, and generate configuration model graphs corresponding to all four networks and compute their S, $\langle d \rangle$ and $\langle C \rangle$. Are the results expected or are they surprising?


T1
S1
T2
T1
S1
S2
T2