DEC 2019

ORACLE LABS STUDENT PROGRAM

GRAPH ALGORITHMS TASK

Prepared by: Mohamed Jalal Maaouni

CHOICE OF LANGUAGE

I opted to code this project in C. I chose C to force myself to recode all necessary data structures i needed for this project so I would avoid introducing any overhead. It is also to have total control of memory management and garbage collection.

The task stated that one should actually implement the algorithm; thus, i judged it being in my interest to start at as low a level of abstraction as possible.

Making this choice meant having to give up the object oriented paradigm. However, the induced verbosity, i hope, is not a major drawback of my code's readability.

TASK 1 SCC

Major points:

- Input
- Implementation of Kosaraju's Algorithm.
- Classification of strongly connected components.
- Implementation of The forward backward algorithm as a parallel alternative to finding SCC.

Input

Since the formatting of the input can be of our devise; I selected the following format:

- first line contains the number of the vertices **n** and the number of edges **m**.
- Then follow **m** lines representing the directed edges.

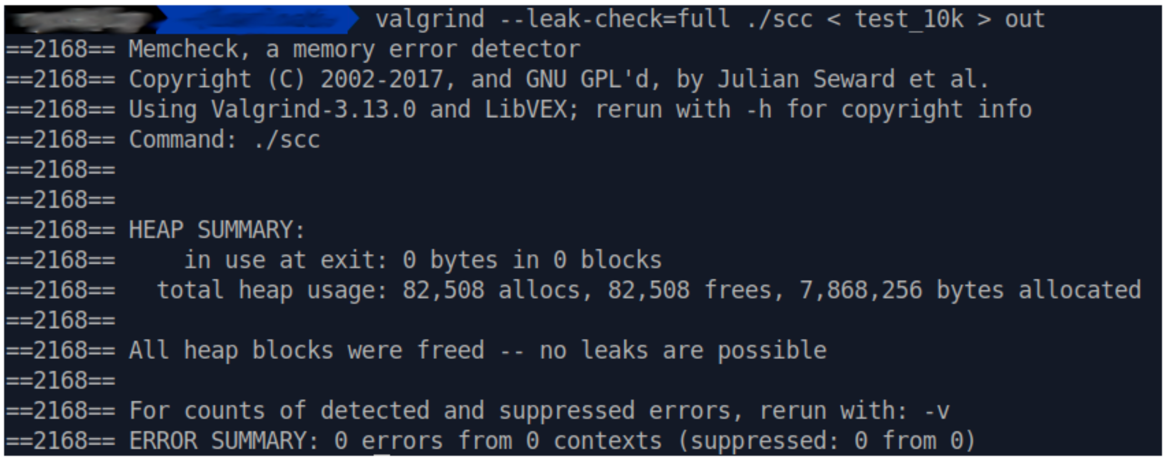
Error management is at a minimum, I only check if the vertices in the given edges belong to the vertex range.

Kosaraju's Algorithm

- Time complexity : O(V+E)
  The time complexity of this algorithm is linear and matches the lower bound. However, it is slower, in practice, than Tarjan's algorithm yet i chose the former for its elegance and readability.

- Memory consumption

  implementing extendable vectors for the adjacency list allows for minimal memory consumption.
  implementing a stack structure allows for direct freeing of every layer when it is popped.
  the next image is a Valgrind test of the program with a graph of 10000 vertices , 2499 edges and 5000 sccs. All memory allocated was freed.



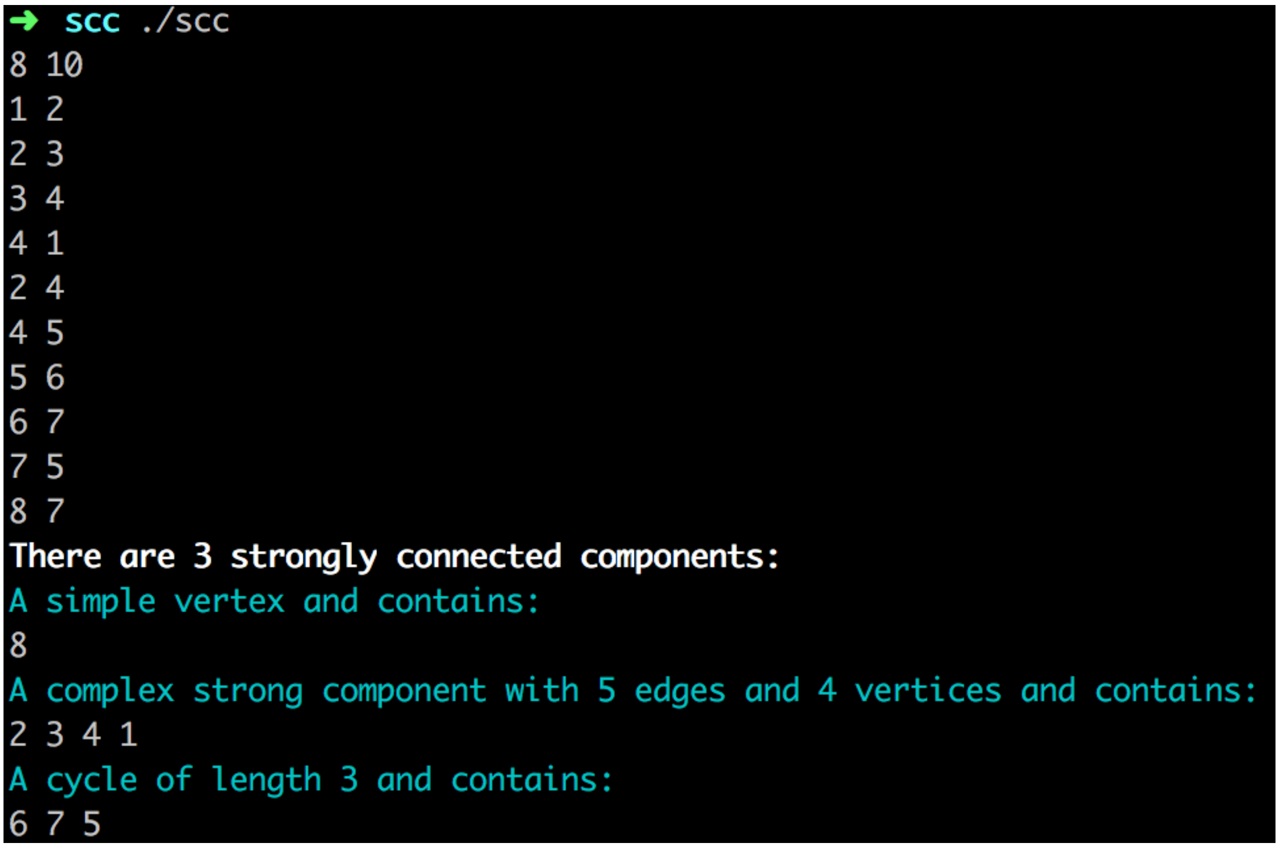
```
[redacted] valgrind --leak-check=full ./scc < test_10k > out
==2168== Memcheck, a memory error detector
==2168== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2168== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2168== Command: ./scc
==2168==
==2168==
==2168== HEAP SUMMARY:
==2168==     in use at exit: 0 bytes in 0 blocks
==2168==   total heap usage: 82,508 allocs, 82,508 frees, 7,868,256 bytes allocated
==2168==
==2168== All heap blocks were freed -- no leaks are possible
==2168==
==2168== For counts of detected and suppressed errors, rerun with: -v
==2168== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Classification of SCCs

The program classifies each scc into three groups

- complex scc with # of edges larger than # of vertices
- cycle
- simple vertex.

Here is an example with 8 vertices and 10 edges.



```
➜  scc ./scc
8 10
1 2
2 3
3 4
4 1
2 4
4 5
5 6
6 7
7 5
8 7
There are 3 strongly connected components:
A simple vertex and contains:
8
A complex strong component with 5 edges and 4 vertices and contains:
2 3 4 1
A cycle of length 3 and contains:
6 7 5
```

The Forward Backward Algorithm

- Difficulty to parallelize DFS based algorithms

  Both Tarjan and Kosaraju algorithms are hard to parallelize because they rely on DFS and DFS sequentially generates only one subproblem and thus renders parallel computing almost impossible.

- Alternative

  In order to benefit from parallel computing, one needs to work within the paradigm of Divide and Conquer.

  One way to do this is to choose a pivot and find all points that can be reached from this pivot in the graph and all points that can be reached from the pivot in the transpose graph.

  The interersection of these two sets will constitute a scc with the pivot in it and will produce 3 independent subproblems.

The 3 subproblems are:

- nodes reachable from the pivot in the graph but not in the transpose graph.
- nodes reachable from the pivot in the graph but not in the transpose graph.
- nodes not reachable from the pivot in neither graph.

A thread will then be dedicated to execute each subproblem.

- Issues

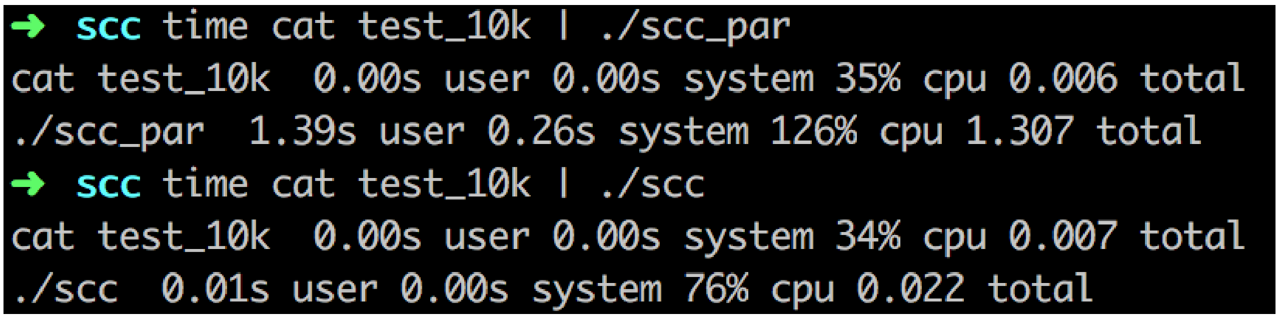
  Although, I tried to use only one memory area accessible to all thread and used MUTEX to synchronize memory access of threads. I could not find a way around copying the vertices that will be part of each subproblem and this constitutes the bottleneck of my implementation. Surpassing this issue may result in considerable performance improvement

- Positive points

  This implementation may allow to find the scc of a vertex without having to traverse the whole graph. In fact, the program prints sccs as they are found and starts printing before the sequential implem.

- Conclusion

  Testing shows that the parallel implementation is orders of magnitude slower than the sequential one. However, it takes better advantage of the CPU. And it can be optimal to find the scc of a single vertex if its size is significantly smaller than the whole graph. Additionally, solving the transition bottleneck coupled with a total parallelization of the implementation (parallel sort, parallel search…) could lead to better performance. Unfortunately, the tight deadline did not allow me to go further in this path.



```
➜  scc time cat test_10k | ./scc_par
cat test_10k  0.00s user 0.00s system 35% cpu 0.006 total
./scc_par  1.39s user 0.26s system 126% cpu 1.307 total
➜  scc time cat test_10k | ./scc
cat test_10k  0.00s user 0.00s system 34% cpu 0.007 total
./scc  0.01s user 0.00s system 76% cpu 0.022 total
```

Applications

- 2SAT

  This is a standard application of sccs. Given, a boolean expression which is a conjunction of disjunctions, we could check if it satisfiable in linear time when a brute force solution would take exponential time. We achieve this result by constructing a graph of **2 * N** vertices where **N** is the number of boolean variables in the expression with edges corresponding to logical relations.

- Condensation graph

  Given a graph, we could compress it by making each vertex in the condensation graph correspond to an scc in the original graph. The resulting graph is also acyclic.

TASK 2 PATH FINDING

Major points:

- Input
- Generation of all paths
- Exponential growth of the output
- Speculative link to the next task

Input

- first line contains the number of the vertices **n,** the number of edges **m**, **1** or **0** depending on whether the graph is directed or not and **x** the given vertex.
- Then follow **m** lines representing the directed/undirected edges.

Error management is at a minimum, I only check if the vertices in the given edges belong to the vertex range.

Generation of all paths

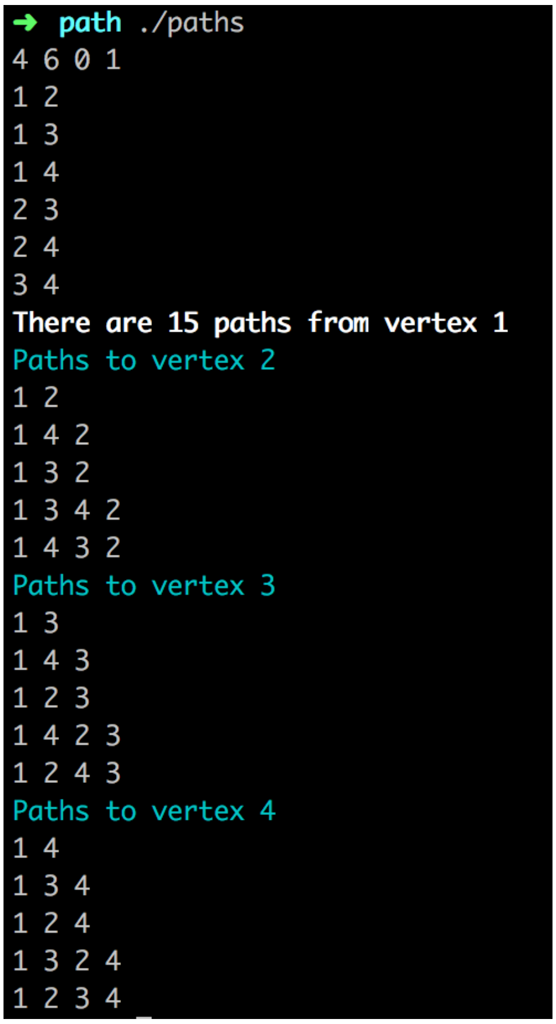
I worked under the assumption that a path is a list of connected edges where each vertex only appears once.

I traversed the graph in a bfs manner and pushed the spurs i obtained thus far to a queue. At each queue pop, i would loop over the adjacency list of the last vertex and if does not appear in the spur it is appended to it and then added to the queue.

This process continues until the queue is emptied.

Lastly the paths obtained are sorted by last vertex then by length to provide a pleasant printing.

Example



```
➜  path ./paths
4 6 0 1
1 2
1 3
1 4
2 3
2 4
3 4
There are 15 paths from vertex 1
Paths to vertex 2
1 2
1 4 2
1 3 2
1 3 4 2
1 4 3 2
Paths to vertex 3
1 3
1 4 3
1 2 3
1 4 2 3
1 2 4 3
Paths to vertex 4
1 4
1 3 4
1 2 4
1 3 2 4
1 2 3 4
```

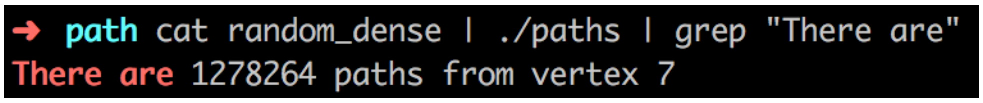
Exponential growth of the output

the output grows exponentially with every edge added to the graph.

If the graph is fully connected, we would get **O(v!)** paths where v is the number of vertices.

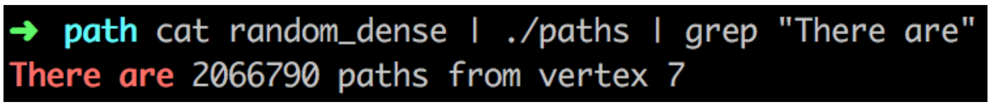
The following is the output of the program on a random graph of a 100 vertices from vertex 7.

for 100 random edges we get:



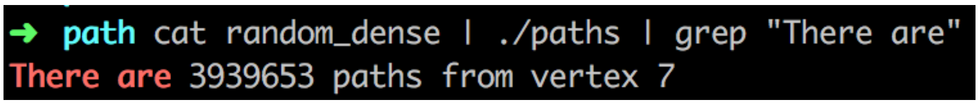
```
➜ path cat random_dense | ./paths | grep "There are"
There are 1278264 paths from vertex 7
```

after adding 2 additional random edges:



```
➜ path cat random_dense | ./paths | grep "There are"
There are 2066790 paths from vertex 7
```

after adding 2 additional random edges:



```
➜ path cat random_dense | ./paths | grep "There are"
There are 3939653 paths from vertex 7
```

Since the problem is NP-complete, there is no efficient way to resolve this task.

Speculative link to the next task

I really had a hard time comprehending this question and thought that I must be misunderstanding something. I concluded that one of the applications of this task is to be used in the following one.

Thus, I will rely on this program to test for possibility that two vertices will have the same rank.

TASK 3 RANKING

Major points:

- Input
- Proposed Solution
- Potential for parallelization
- Applications

Input

- first line contains the number of the vertices **n,** the number of edges **m**, and **x** the vertex where to start the greedy search.
- Then follow **m** lines representing the directed/undirected edges.

Error management is at a minimum, I only check if the vertices in the given edges belong to the vertex range.

Proposed Solution

Vertex ranking is NP-hard for most graphs. While researching this problem, I only stumbled upon papers that deal with trees or specific types of graphs.

Consequently, I only managed to provide a program that provides an upper bound for the vertex ranking.

The implementation is based on the previous task, it starts from a given vertex, gives it 1 as a rank and proceeds to its neighbors and assigns minimal ranks to them (Here too, it is obvious that only a fraction of the search space is explored). The implementation always produces a valid ranking though not necessarily minimal.

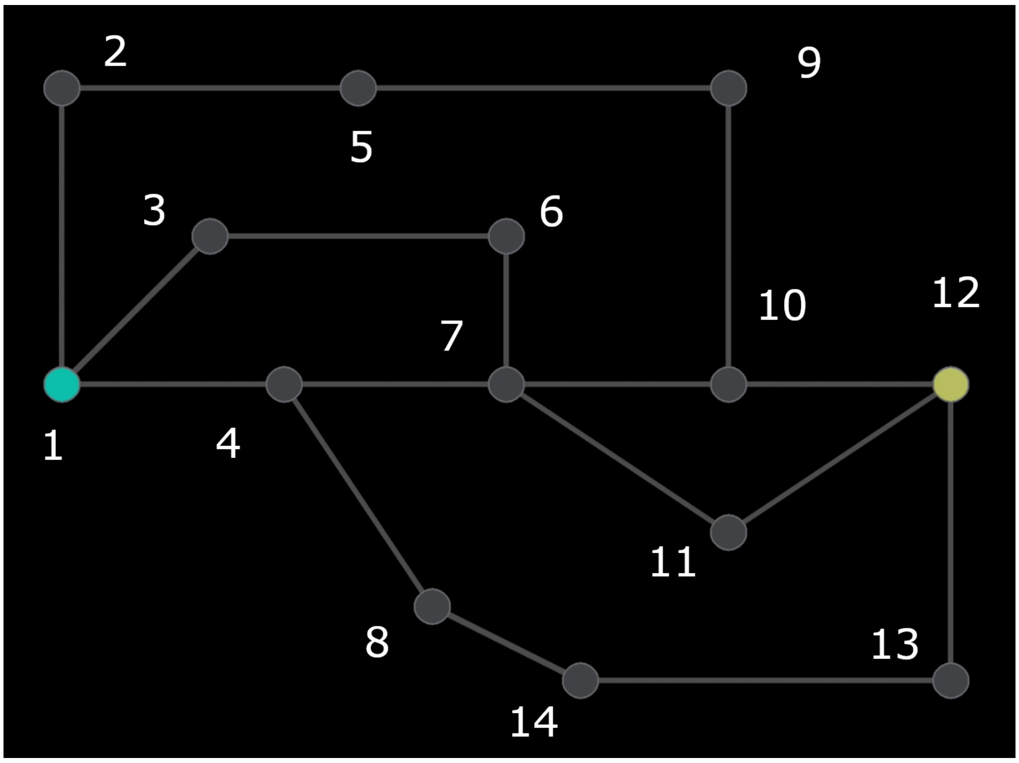
In every transition to vertex i, a check is done on rank values from 1 to the number of vertices in the graph.

For each rank value, all paths to all vertices that were assigned the value are checked for a value greater than the rank value or for a vertex still not assigned. If the check fails, the following rank value is checked.

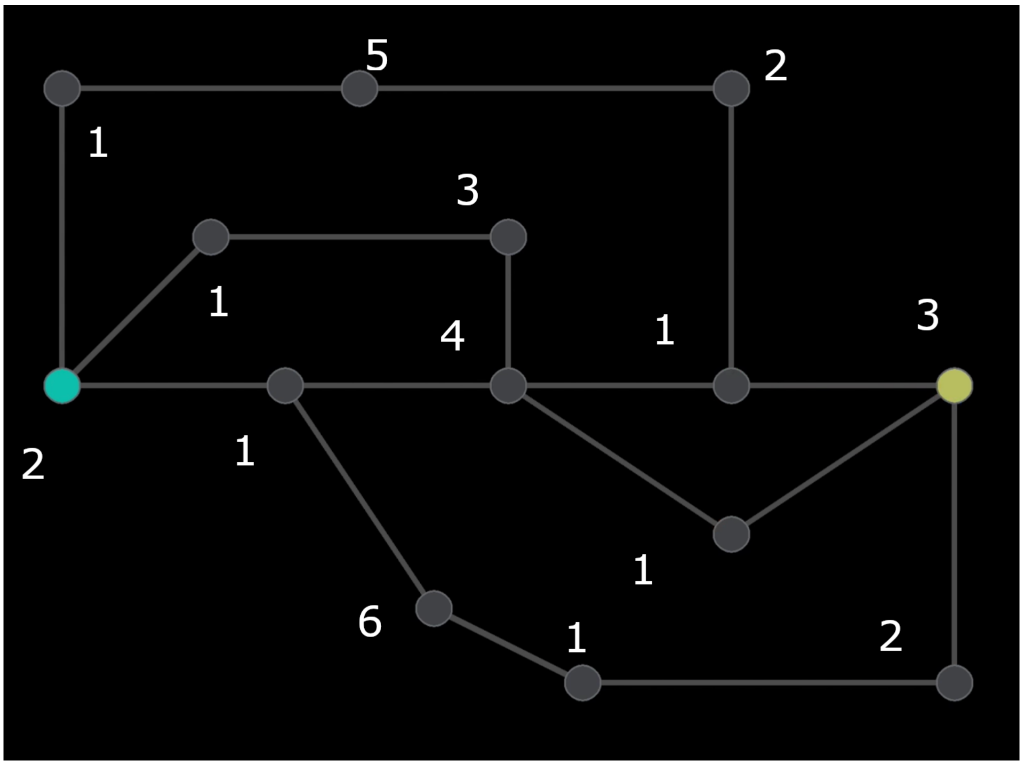
Example

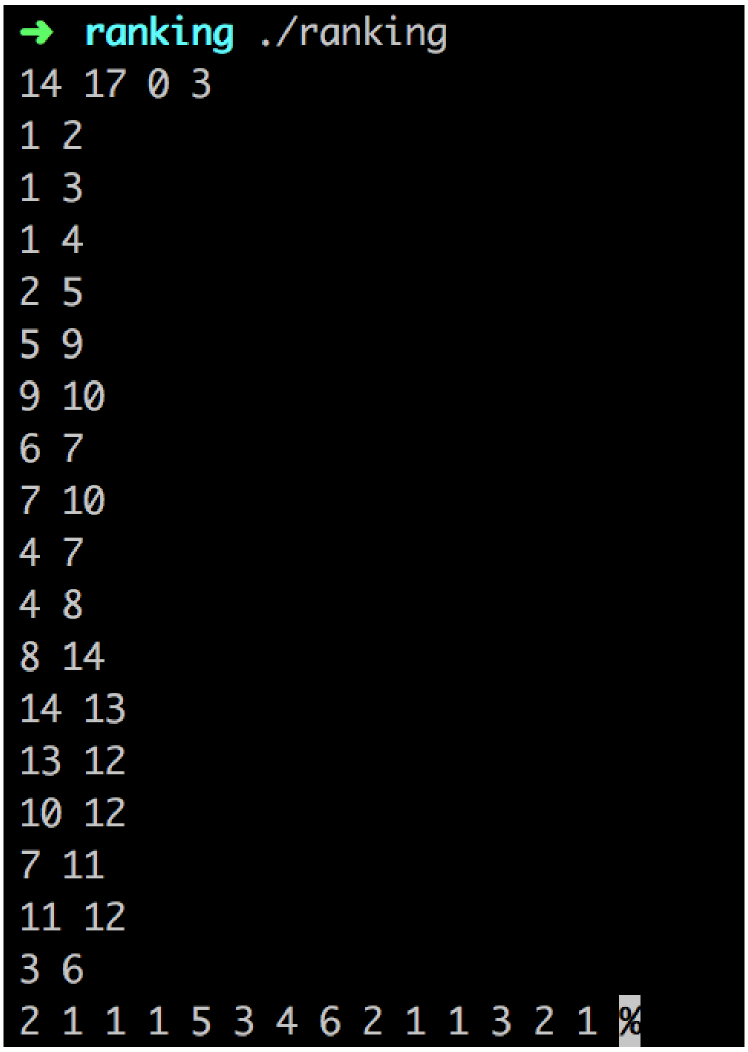
Add a little bit of body text

Consider the following undirected graph with 14 vertex and 17 edges.



Here is a possible ranking of order 6.





```
➜  ranking ./ranking
14 17 0 3
1 2
1 3
1 4
2 5
5 9
9 10
6 7
7 10
4 7
4 8
8 14
14 13
13 12
10 12
7 11
11 12
3 6
2 1 1 1 5 3 4 6 2 1 1 3 2 1 %
```

Potential for parallelization

Since at every transition, the choice of the neighbor to explore next totally changes the whole state. All states should be equiprobable to be explored.

Thus, it is possible parallelize the subsequent computations and choose the optimal one and backtrack from there.

Applications

A vertex ranking provides an ordering of vertices. Thus if one has certain events that could only processed after others have been finished, one could create a graph where the edges would represent the constraints.

This can be used in assembly of modular products or in the processing of parallel queries.