Comp 4905 project final report

Title: an analysis and implementation of sublinear degree+1 graph colouring.

Name: Arthur Morris

Supervisor: Dr. Evangelos Kranakis

Acknowledgements – Sepehr Assadi, Yu Chen, and Sanjeev Khanna, their paper is the theoretical basis of this one, none of this would be possible without their work.

Abstract:

Improving the efficiency of algorithms is a key objective of algorithm development. One of the ways to make an algorithm more efficient is to parallelize it. One of the techniques often used for effective parallelization is graph colouring, grouping all the vertexes of a graph so that there is no edge connecting any two vertexes in the same group. This is used in a number of applications, mostly to prevent conflicts for resources. Graph colouring can also be used to break symmetry in distributed systems where vertexes represent compute units and edges encode the network. Until recently this was done using a simple linear time greedy algorithm. Recently, a new sublinear algorithm was proven for degree+1 colours, but no implementation of this is available yet. This new algorithm uses a fundamentally different approach. The Greedy algorithm checks every neighbor then assigns the first available colour. The new algorithm limits the possible colours at each vertex, and then uses this to eliminate irrelevant edges. The goal of this paper is to produce and analyze an implementation of this algorithm and some adjustments of it using c++.

Background material

Graph colouring is a fundamental problem in graph theory. Graph theory is a branch of mathematics that deals with abstract graphs. Graphs are mathematical structures that consist of vertices and edges. Each edge connects two different vertices together. There are several types of graph colouring. This paper is about vertex colouring, the process of assigning a number or colour to each vertex so that no edge has the same colour on both ends. For the purposes of this paper it is more natural to think of it a different way. Graph colouring can be thought of as the process of splitting the graph up into subgraphs such that each subgraph has no edges.

Any graph can be coloured in at most $\Delta + 1$ colours, where Δ is the degree or number of edges of the densest vertex. In $\Delta + 1$ colouring, any valid partial colouring can be extended into a full colouring by assigning colours to each of the uncoloured vertexes. This has various uses both practical and theoretical. It is used to split up graphs into an optimal, or near optimal, number of sets. The simplest example of this is process scheduling to avoid resource conflicts in a parallel context. To do this, one constructs a graph with a vertex for each task. Then edges are added so all of the processes that require each resource are all connected together. Then the graph is coloured, each colour is a set of non-conflicting tasks that can be done concurrently. This process can add significant overhead so improving graph colouring is an important practical problem. In the theoretical world, a lot of problems can be reduced to graph colouring or have reductions that involve graph colouring, so having a sublinear algorithm for graph colouring opens up sublinear solutions to many other problems.

Position in the existing literature

Assadi, Chen and Khanna [1] released a sub linear time algorithm for $\Delta + 1$ colouring. We will refer to this as the ACK algorithm. $\Delta + 1$ colouring uses only $\Delta + 1$ colours, this is the smallest possible number of colours needed to colour an abstract graph. This new algorithm can be used on dynamic streams or fixed graphs in the query model. It can also be used in the standard or MPC paradigm. They accomplish this by using a fundamentally different method. Random sampling is used to sparsify the graph to reduce the number of edges, thus reducing the work required to colour the graph. There have been several improvements recently to MPC and stream colouring. But none in normal graph colouring. This new algorithm is the first major advancement to graph colouring in the general model ever, finally replacing the greedy algorithms. The greedy algorithm is so old and fundamental to graph theory that there is no record of who or when it was first proposed. In this paper implement the new ACK algorithm and compare it to the classic greedy algorithm.

scope and limitations

Assadi, Chen and Khanna [1] proved their new algorithm in three paradigms: streaming, query and massively parallel (MPC). In this paper we will only consider the query model, because it is the simplest. We avoid making alterations to the basic algorithm that might require significant additional proof.

Overview of the Assadi, Chen and Khanna (ACK) algorithm:

Assadi et al describe their algorithm in the following way:

> “After sampling O (log(n)) colors for each vertex randomly, the total number of monochromatic edges is only O (n $\cdot \log_2(n)$). With high probability …, by computing a proper coloring of G using only these O (n $\cdot \log_2(n)$). edges … we obtain a (Δ + 1) coloring of G.”
>
> (Assadi, Chen, Khanna, 2018: section 1.1)

Or more formally:

> “ColoringAlgorithm (G,Δ): A meta-algorithm for finding a (Δ+1)-coloring in a graph G(V, E) with maximum degree Δ.
>
> 1. Sample Θ (log (n)) colors L(v) uniformly at random for each vertex v ∈ V (as in Theorem 1).

2. Define, for each color c ∈ [Δ + 1], a set χc ⊆ V where v ∈ χc iff c ∈ L(v).

3. Define E conflict as the set of all edges (u, v) where both u, v ∈ χc for some c ∈ [Δ + 1].

4. Construct the conflict graph G conflict (V, E conflict).

5. Find a proper list-coloring of G conflict (V, E conflict) with L(v) being the color list of vertex v ∈ V.”

(1: section 4)

There are three basic steps:

1. Assign log n colours to each vertex, keeping track of the lists of vertexes that get each colour.
2. Use these lists to find the monochromatic or conflict edges.
3. Perform a greedy colouring using only the sampled colours and the conflict edges.

Assadi et al have proven that this runs in $\tilde{O}(n\sqrt{n})$. (1: section 1.1) this means it runs in $O(n\sqrt{n}\,\log_c n)$ for some integer c (2: Standard Notation). The choice of this c is critical for any practical implementation. There is a upper bound on it, if it is too big then the number of colours sampled will be too small, so the probability that each vertex does not sample a valid colour gets higher, resulting in uncoloured vertices. But the smaller it is the slower the algorithm runs. Assadi et al did not specify how to choose this constant. Finding this constant was one of our goals of this paper. Note that the actual running algorithm is very different from the proof.

As we discuss below we have shown that this can be improved on by adding this fourth step:

4. Greedy colour any remaining vertexes

This extra step catches any uncoloured “orphan” vertexes that were extremely unlucky and did not sample a valid colour. This new step eliminates the possibility that the algorithm does not produce a valid colouring. The expected number of remaining “orphan” vertexes is 0. The actual number is dependent on the exact number of colours sampled. Assadi et al did not give an exact number, only an O bound for how many to sample. If the number of “orphan” vertexes is less than $\sqrt{n}$, adding this step does not affect the O bound on the runtime. This hybrid algorithm has a different value for the constant c, it has an optimal value that balances the two algorithms. If it is off in either direction it runs slower.

Pseudocode:

If this is unclear see the actual code in the appendix.

```
ColoringAlgorithm (V,N,E,Δ):

    *Step 1:

    For each vertex in V:  *n

        For each c in (0, log N): *log n

            Vertex.colour [c] = random in (0, Δ)

            Sets[Vertex.colour [c]].add Vertex

    *Time for this step : n log n, total time: n log n


    *Step 2:

    For each set in sets:   *Δ

        For each vertex in set: *n/Δ

            For each vertex2 in set where vertex2 > vertex *n/Δ

                If E(vertex, vertex2):

                    Vertex.conflict.add vertex2

                    Vertex2.conflict.add vertex

    *Time for this step : $\frac{n^2}{\Delta}$ this is O($n\sqrt{n}$) for $\Delta > \sqrt{n}$, total time: $\frac{n^2}{\Delta}$ + n log n


    *Step 3:
```

```
Vertexes = V
For each run in (0, log n): * log n
    For each vertex in Vertexes where vertex.finalcolour = null: *n
        For each vertex2 in Vertex.conflict: *$log^2 n$
            If vertex2.finalcolour = vertex.colour [run]
                Nextset.add vertex
                break to next vertex
        vertex.finalcolour = vertex.colour [run]
    Vertexes = Nextset
```

*Time for this step : $n\ log^3 n$, total time: $\frac{n^2}{\Delta}$ + n log n + $n\ log^3 n$

*step 4:

```
For each vertex in Vertexes
    If vertex.finalcolour = null
        For each vertex2 in Vertexes
            E(vertex, vertex2):
```

*Time for this step o * n where o is the count of vertices where vertex.finalcolour = null.

Total time: $\frac{n^2}{\Delta}$ + n log n + $n\ log^3 n$ + (o * n)

* We do not have a way to predict o, but we know there is a positive correlation between o and the log base c. if c has been chosen correctly the this step does not add additional time since the expected number if vertexes where vertex.finalcolour = null is

0. We observed that it was more efficient to intentionally undersize the sample than fix the resulting orphans then use a correctly sized sample.

Proof:

The following is only a rough outline of the proof for details see the paper Assadi, Chen, Khanna, 2018 for a complete proof.

The proof hinges on showing that log n colours per vertex is sufficient to colour the graph. To prove this Assadi et al split the graph into two parts and prove them separately. To split the graph they used a Harris Schneider Su (HSS) decomposition. [3] (1: section 2.1) The HSS decomposition is based on triangles, or neighbors common to a pair of vertexes. It uses the number of these to split the vertexes into dense and sparse sets according to a density ε. If v is a ε-sparse vertex then at least $\varepsilon \cdot \Delta$ of its neighbors have at most $(1-\varepsilon) \cdot \Delta$ neighbors in common with v. (1: section 2.2) Any vertex that is not sparse is dense. Assadi et al then proved the bound for ε-sparse and dense vertexes separately.

Assadi et al used standard graph theory to show that log n colours is sufficient for the ε-sparse vertexes. To do this they started with the non-edges of each ε-sparse vertex, each vertex has at least "$\varepsilon^2 * \binom{\Delta}{2}$ non edges". (1: section A.1) They then find and count the anti-edges where both ends are assigned the same colour. They then found a bound for the number of these edges. Lastly they summed over all ε-sparse vertexes.

Meanwhile for the dense vertexes Assadi et al used a colorful matching. First they broke the dense vertexes up into cliques where each connected component of v-dense is a clique.

“For each color, if we can find a pair of vertices u, v such that (u, v) is not in G conflict, u and v are not in the colorful matching yet, and L(u) and L(v) both contain the same color, then we add (u, v) with this color to the colorful matching.“ (1: section 4.11)

Lastly they used this to construct a pallet graph using the Hopcraft Karp algorithm for bipartite matching.(1: section 4.11)(4)

All of this shows that with high probability $\log_c n$ colours is sufficient to colour the graph. High probability means p > $\frac{1}{poly\ n}$ for some increasing polynomial.

Overview of the greedy algorithm:

The algorithm works as follows:

Check every neighbor of each vertex then assign it one of the unused colours. This obviously takes O (nΔ) time.

Pseudocode:

If this is unclear see the actual code in the appendix.

```
For each vert in vertexes
    colours [] = new colour [Δ]
    For each neigh in vert.neighbor
        If neigh.Colour != null
            colours[neigh.Colour]=1
```

```
For each c in colours
    If c == null
        Vert.colour = c
```

Methodology

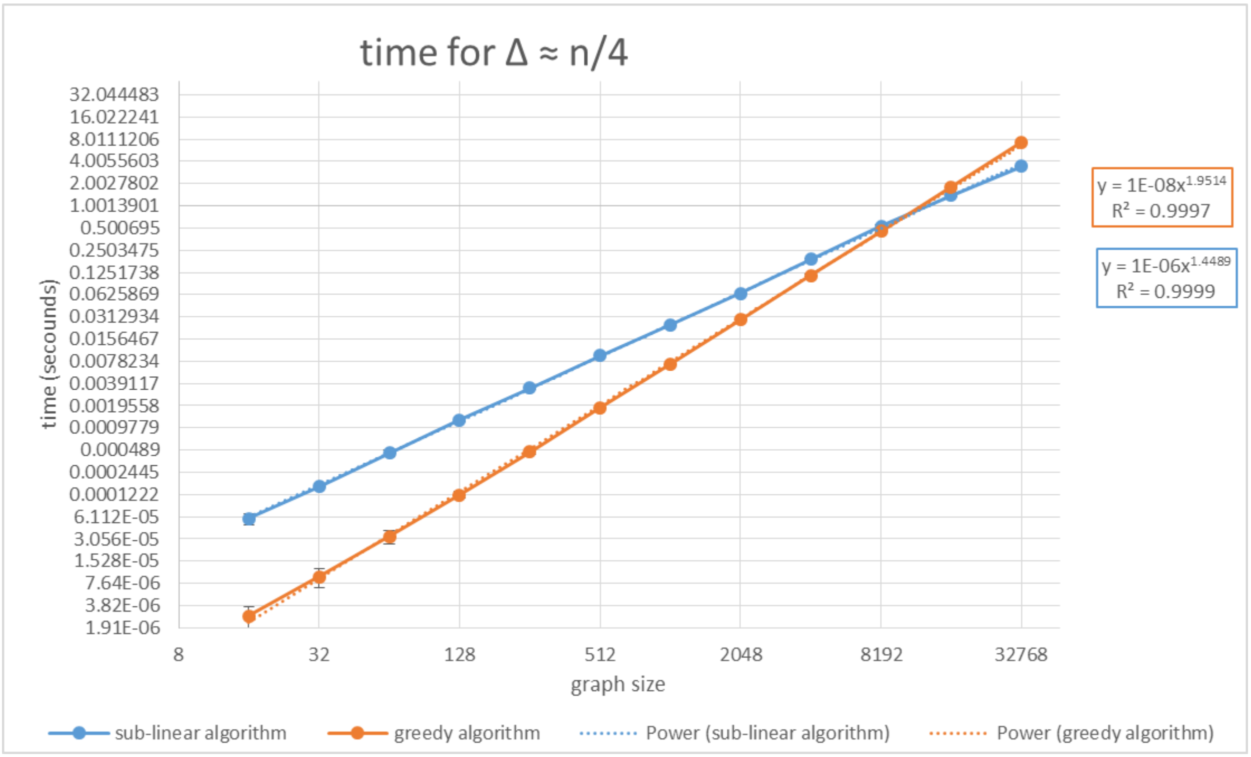
We implemented both algorithms and tested them on randomly generated graphs in the size range 16 < n < 32768, increasing in powers of 2. All the testing was done in C++. Our code is available in the appendix. We used a linux desktop system to run the tests. The clock_t function from the standard file time.h was used for all the timing. This works by counting CPU cycles so it gives very accurate timing even for very short runs. Only the actual run is timed. The algorithms where tested on random graphs. These where generated by starting with a graph with no edges then adding each edge based on a random value. So for example in the n/2 dense graph each edge has a 50% probability of existing. The query model uses $O(n^2)$ memory so this was a large as possible with the memory available. For the time comparison 250 tests where run on each data point, 25 runs each for 10 different graphs. For the other comparisons only 25 runs on a single graph where run.

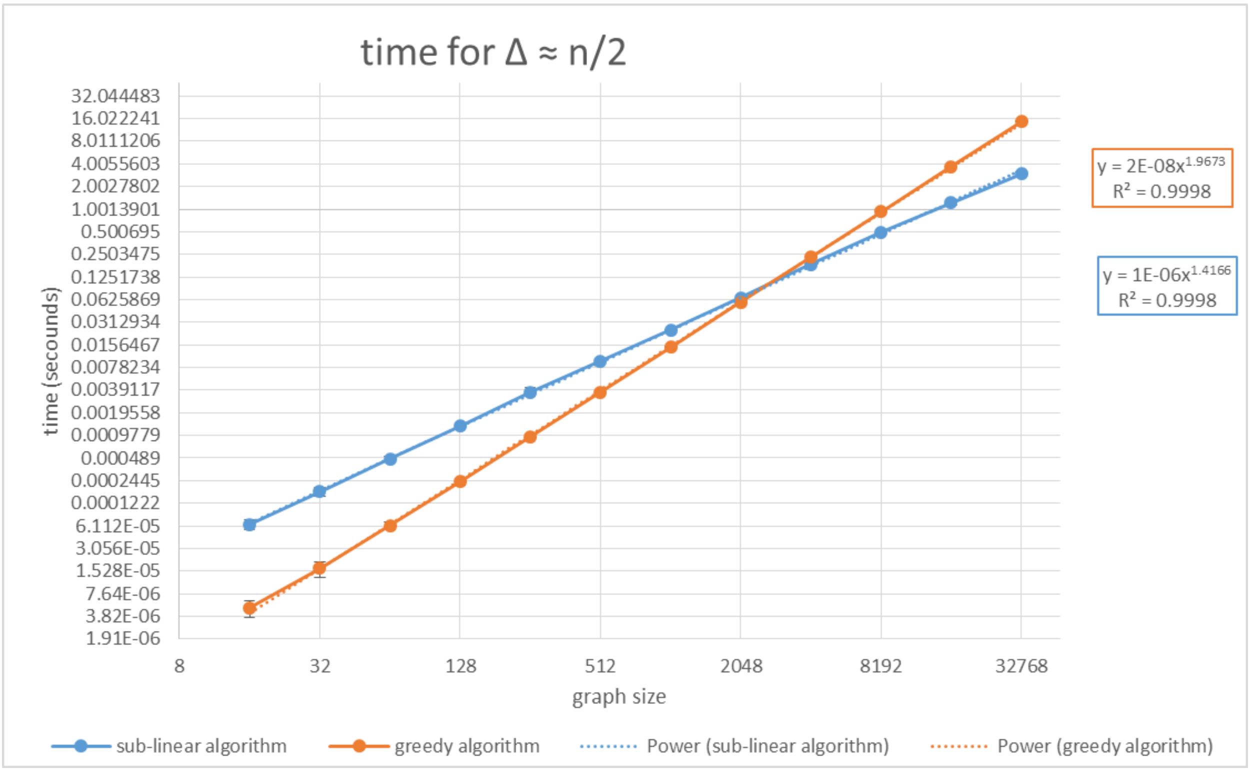
Results

O bound

For this test $\log_2 n$ colours per vertex where sampled. This was chosen as a compromise between colouring all the vertices and speed. This was tested on three graph densities, n/4, n/2, 3n/4. The results are plotted in fig 1, 2, 3. The slopes of the lines of best fit are almost exactly what was expected. Greedy colouring is O (nΔ), this is O (n^2) if Δ grows linearly with n. The sublinear algorithm is O ($n\sqrt{n}$), or O ($n^{\sqrt{2}}$).

We can see exactly what was expected in the graphs. The greedy colouring time increased as density increases. The sublinear algorithm gets faster as density increases because it has more colours. The powers are almost exactly as the theory expected them to be, the line of best fit did not account for the log factor, so the values are a bit skewed but not anywhere near enough to make a difference. All of the r^2 values are extremely high, this means that the lines of best fit are fitted almost exactly to the data, this was expected given the linear nature of the algorithm and the accuracy of the timing.


time for Δ ≈ n/4
time (secounds)
32.044483
16.022241
8.0111206
4.0055603
2.0027802
1.0013901
0.500695
0.2503475
0.1251738
0.0625869
0.0312934
0.0156467
0.0078234
0.0039117
0.0019558
0.0009779
0.000489
0.0002445
0.0001222
6.112E-05
3.056E-05
1.528E-05
7.64E-06
3.82E-06
1.91E-06
8
32
128
512
2048
8192
32768
graph size
y = 1E-08x^1.9514
R² = 0.9997
y = 1E-06x^1.4489
R² = 0.9999
sub-linear algorithm
greedy algorithm
Power (sub-linear algorithm)
Power (greedy algorithm)


time for Δ ≈ n/2
time (secounds)
32.044483
16.022241
8.0111206
4.0055603
2.0027802
1.0013901
0.500695
0.2503475
0.1251738
0.0625869
0.0312934
0.0156467
0.0078234
0.0039117
0.0019558
0.0009779
0.000489
0.0002445
0.0001222
6.112E-05
3.056E-05
1.528E-05
7.64E-06
3.82E-06
1.91E-06
8
32
128
512
2048
8192
32768
graph size
y = 2E-08x^1.9673
R² = 0.9998
y = 1E-06x^1.4166
R² = 0.9998
sub-linear algorithm
greedy algorithm
Power (sub-linear algorithm)
Power (greedy algorithm)

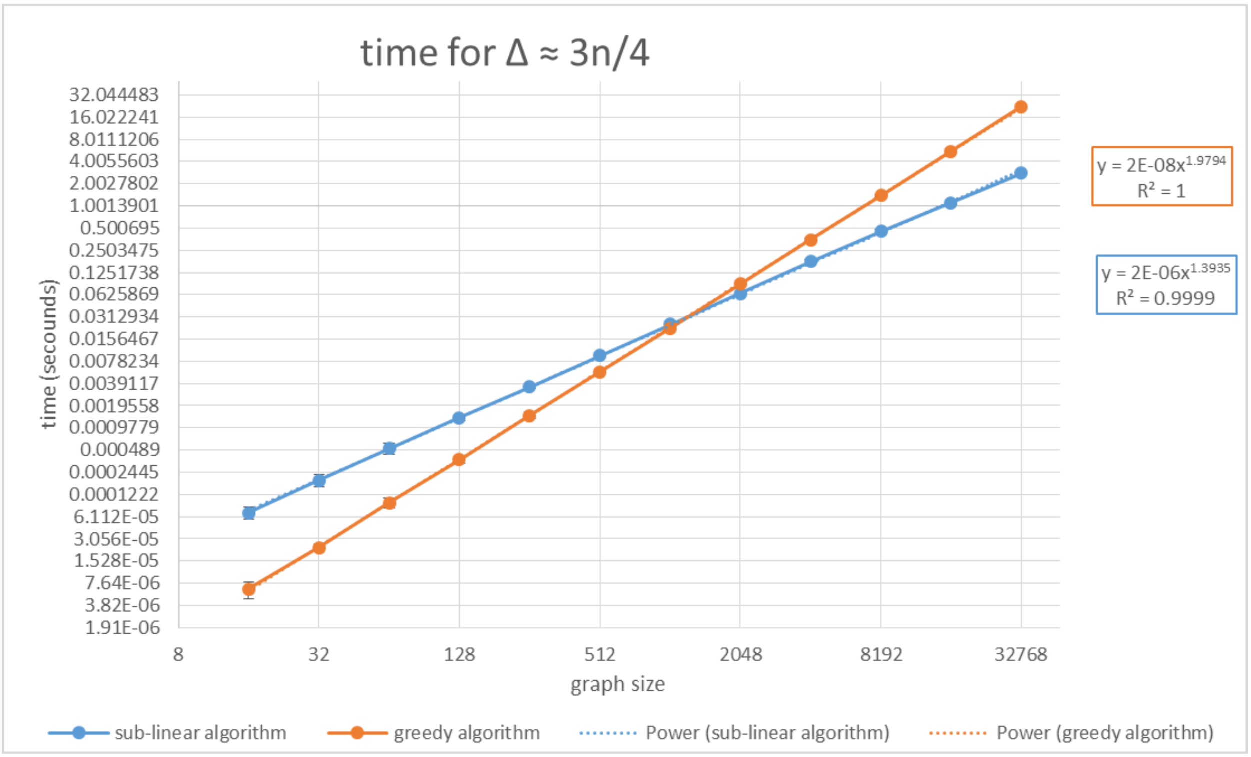


Fig 1,2,3

These graphs all show the time against graph size for the ACK and greedy algorithms with lines of best fit. Note these are log log plots.

the optimal value for c

The smallest number of colours sampled that consistently coloured all the vertexes across all the tested graph sizes was $3 * \log_2 n$. I.e. c = $\sqrt[3]{2}$. This was so slow it is not practical to use. Theoretically it is faster than the greedy algorithm but the crossover point of the lines of best fit is n = 794,278, taking an estimated 8092 seconds or 2 hours 14 minutes and 52 seconds per trial. This was not attempted. Note that this test was only done on random graphs, different graph structures may affect the optimal c value.

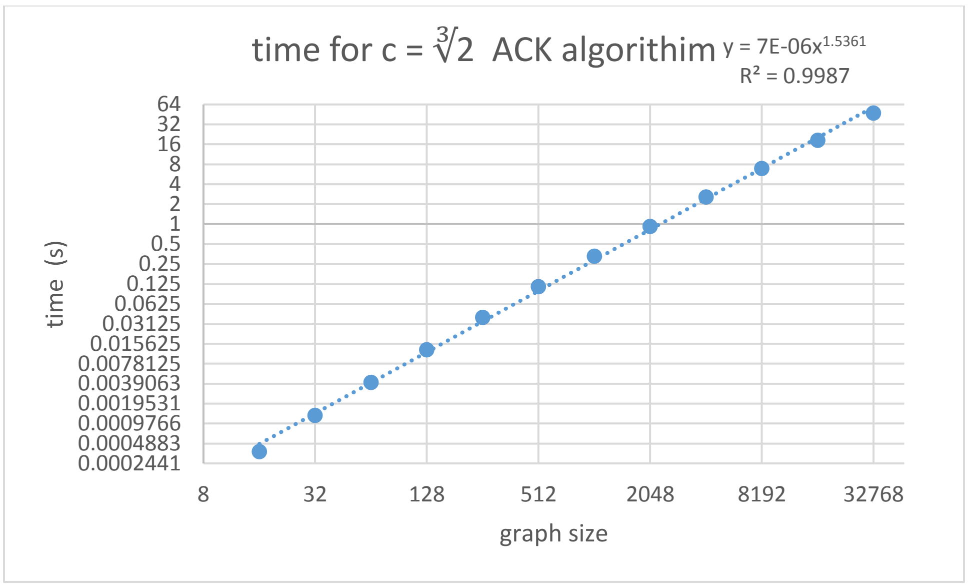


Fig 4

This graph shows the time against graph size for the ACK algorithm. c = $\sqrt[3]{2}$. With a line of best fit. Note this is a log log plot.

The hybrid algorithm

The hybrid algorithm uses the sublinear algorithm but intentionally under samples the colours so not every vertex receives a valid colour, this makes the sparsifyed graph much smaller meaning it runs faster. It then greedy colours the remaining colourless vertexes. We can see in fig 6 that this grows proportionally to $n^{\sqrt{2}}$. Comparing this to fig 4, we find that both the power and the linear factor are smaller so the hybrid algorithm is faster than the sublinear algorithm.

They have the same O bound. While I can't actually prove this is faster mathematically, the experimental results strongly imply that it is.

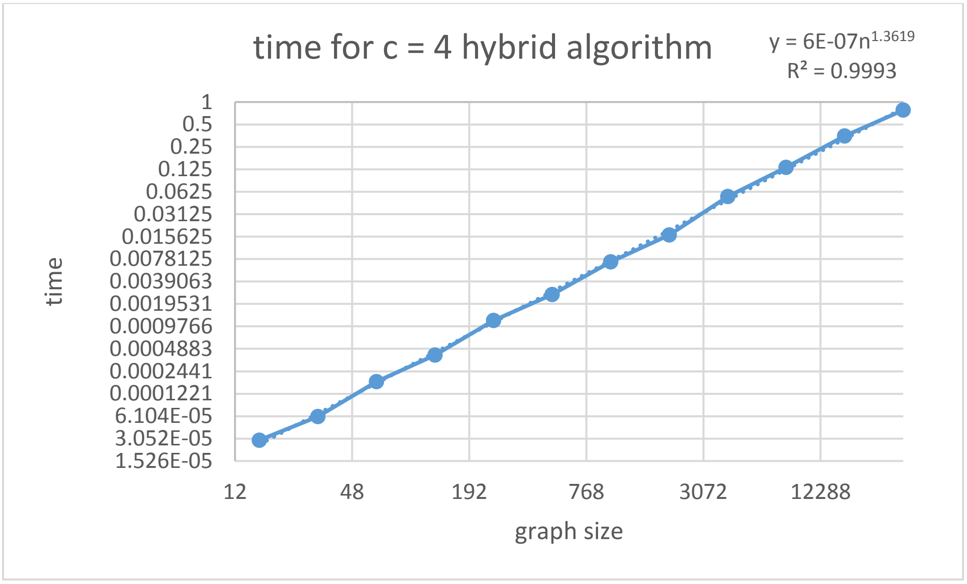


Fig 5

This graph shows the time against graph size for the hybrid algorithm. c = 4. With a line of best fit. Note this is a log log plot.

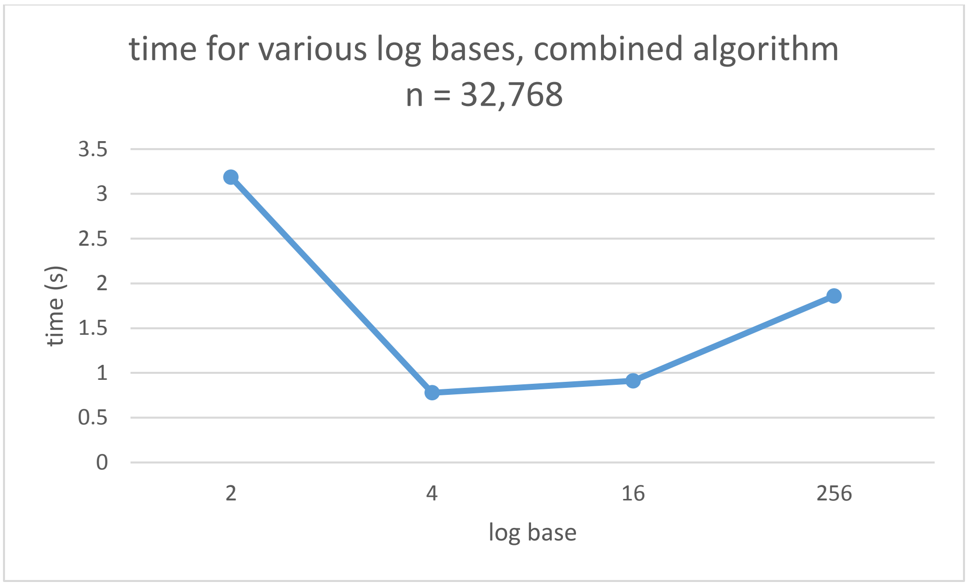


Fig 6 this shows the time taken to colour 32,768 vertices as we increase the log base, shrinking the sample size.

Conclusions

Result summary

We have demonstrated that Assadi, Chen and Khanna's sublinear graph colouring algorithm works in theory but in reality it is only faster than the greedy algorithm for unrealistically large graphs. However with some simple tweaks it can be made much faster. By under sampling and then greedy colouring the orphan vertices we can colour the graph substantially faster the greedy algorithm. We believe this hybrid algorithm has the same O bound on its run time, it is also sublinear.

Recommendations

The ACK algorithm is not quite ready to replace the traditional greedy algorithm in production applications. The constants on the ACK algorithm as written make it unusable. Our hybrid algorithm is definitely an improvement, it will need to be tested and have the c constant tuned for each application. Our work shows that the improvements are possible and we look forward to the possibilities.

Future work

The ACK graph colouring algorithm is still very new, there are lots of possibilities that have not been tested yet.

The hybrid algorithm changes the constant c optimization problem. It is no longer about finding the largest possible c where all the vertexes sample a valid colour, it becomes a balancing act between the two parts of the hybrid. If c is too low the sublinear algorithm's sparsifyed subgraph is too big and it runs slower. But if c is too big the opposite thing happens, too many vertices are greedy coloured also making it slow.

We believe that sampling the colours dynamically in some way would be best, once a vertex finds a valid colour it does not need to consider the colours sampled after that one. This could be accomplished by sampling different numbers of colours depending on some criterion like density, or it could be done by having multiple sampling rounds, only adding colours to vertices that are still uncoloured.

Since the queries to the graph are predetermined, it should be simple to break the graph down into blocks then store these blocks on a disk, only holding a single block in memory at a time. This will allow colouring larger graphs without extreme amounts of memory.

Citations:

1. Sublinear Algorithms for (Δ+1) Vertex Coloring. By: Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Published 2018. available at: arxiv.org/abs/1807.08886

2. Introduction to Property Testing. By Oded Goldreich. Published 2017. Available at: www.wisdom.weizmann.ac.il/~oded/pt-intro.html

3. Distributed (Δ+1)-Coloring in Sublogarithmic Rounds. By: David G. Harris, Johannes Schneider, Hsin-Hao Su. Published 2016. Available at: arxiv.org/abs/1603.01486

4. J. E. Hopcroft and R. M. Karp. A nˆ5/2 algorithm for maximum matchings in bipartite graphs. In 12th Annual Symposium on Switching and Automata Theory, East Lansing, Michigan, USA, October 13-15, 1971, pages 122–125, 1971. Available at: ieeexplore.ieee.org/document/4569670

Apendix

Code

This was included in case the pseudocode isn’t clear.

Note for this to work on grapghs larger than 1000 vertices run “ulimit -s 100000000” first.

The ACK algorithm:

```
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include <fstream>
#include <vector>
#include <time.h>
#include <sstream>
#include <string>

using namespace std;
class vertex {
public:
  int *colours;
  vector<int> confilict;
  int finalcolour;
};

int main(int argc, char **argv) {
  int n = 0;
  const char *temp[12] = {"16.txt",    "32.txt",    "64.txt",    "128.txt",
                      "256.txt",   "512.txt",   "1024.txt",  "2048.txt",
                      "4096.txt",  "8192.txt",  "16384.txt", "32768.txt"};
for (int x = 0; x < 12; x++) {
  cout << "starting " << temp[x] << "\n";

  ifstream input(temp[x], ifstream::in);

  input >> n;
  int ln = 3*(log2(n) + 1);
  int deg = -1;
  input >> deg;
  srand((int)time(0));
  char edges[n][n];
  for (int i = 0; i < n; i++) {
    for (int c = 0; c < n; c++) {
      input >> edges[i][c];
      edges[i][c] -= '0';
      edges[c][i] = edges[i][c];
    }
  }

  input.close();

  for (int runs = 0; runs < 25; runs++) {
    vector<vertex> vertarr;

    vector<int> sets[deg + 1];
```

```
// initalise vertexs
for (int i = 0; i < n; i++) {
  vertex v;
  vertarr.push_back(v);
  vertarr[i].colours = new int[ln];
  vertarr[i].finalcolour = -1;
}

clock_t start = clock();

// assign random colours
for (int i = 0; i < n; i++) {
  for (int c = 0; c < ln; c++) {
    int colour = rand() % (deg + 1);
    sets[colour].push_back(i);
    vertarr[i].colours[c] = colour;
  }
}
// fill confilct sets
for (int i = 0; i < (deg + 1); i++) {
  for (int c = 0; c < sets[i].size(); c++) {
    for (int t = c; t < sets[i].size(); t++) {
      if (edges[sets[i].at(c)][sets[i].at(t)]) {
        vertarr[sets[i].at(c)].confilict.push_back(sets[i].at(t));
        vertarr[sets[i].at(t)].confilict.push_back(sets[i].at(c));
      }
    }
  }
```

```
}

int x = 0;
for (int cur = 0; cur < ln; cur++) {        // for each colour
  for (int i = 0; i < vertarr.size(); i++) { // for each vertex
    for (int c = 0; c < vertarr[i].confilict.size() && x == 0;c++) { // for each nebour
      if (vertarr[vertarr[i].confilict.at(c)].finalcolour == vertarr[i].colours[cur]) {
        x = 1;
      }
    }
    if (vertarr[i].finalcolour == -1 && x == 0) {
      vertarr[i].finalcolour = vertarr[i].colours[cur];
    }
    x = 0;
  }
}

clock_t end = clock();
double time = (double)(end - start) / CLOCKS_PER_SEC;
int bads = 0;
for (int i = 0; i < n; i++) {
  if (vertarr[i].finalcolour == -1) {
    bads++;
  }
}
for (int i = 0; i < n; i++) {
  if (vertarr[i].finalcolour == -1) {
    cout << i << " sampled";
```

```
          for (int x = 0; x < ln; x++) {
            cout << vertarr[i].colours[x] << " ";
          }
          cout << "\nnebours:";
          for (int t = 0; t < n; t++) {
            if (edges[i][t]) {
              cout << t << " ";
            }
          }
          cout << "\ncolours: ";
          for (int t = 0; t < n; t++) {
            if (edges[i][t]) {
              cout << vertarr[t].finalcolour << " ";
            }
          }
          cout << "\n";
        }
      }
      cout << time << ", ";
      cout << bads << "\n";
    }
  }
  return 0;
}
```

The greedy algorithm:

```
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include <fstream>
#include <vector>
#include <time.h>
using namespace std;
class vertex {
public:
  vector<int> nebours;
  vector<int> colours;
  int finalcolour;
};

int main(int argc, char **argv) {

  char mode = 'n';
  int n = 0;
  const char *temp[12] = {"16.txt",    "32.txt",    "64.txt",    "128.txt",
                         "256.txt",   "512.txt",   "1024.txt",  "2048.txt",
                         "4096.txt",  "8192.txt",  "16384.txt", "32768.txt"};
  for (int x = 0; x < 12; x++) {
```

```
cout << "starting " << temp[x] << "\n";
ifstream input(temp[x], ifstream::in);

input >> n;
int deg = -1;
input >> deg;
srand((int)time(0));
char edges[n][n];

for (int i = 0; i < n; i++) {
  for (int c = 0; c < n; c++) {
    input >> edges[i][c];
    edges[i][c] -= '0';
    edges[c][i] = edges[i][c];
  }
}

input.close();
for (int runs = 0; runs < 25; runs++) {
  vector<vertex> vertarr;

  // initalise vertexs
  for (int i = 0; i < n; i++) {
    vertex v;
    for (int l = 0; l < deg + 1; l++) {
      v.colours.push_back(0);
    }
    v.finalcolour = -1;
    for (int r = 0; r < n; r++) {
      if (edges[i][r]) {
        v.nebours.push_back(r);
      }
    }
    v.finalcolour = -1;
    vertarr.push_back(v);
  }

  clock_t start = clock();
  for (int i = 0; i < n; i++) {
    for (int c = 0; c < vertarr[i].nebours.size(); c++) {
      if (vertarr[vertarr[i].nebours.at(c)].finalcolour != -1) {
        vertarr[i].colours.at(
            vertarr[vertarr[i].nebours.at(c)].finalcolour) = 1;
      }
    }

    for (int c = 0; c < vertarr[i].colours.size(); c++) {
      if (vertarr[i].colours.at(c) == 0) {
        vertarr[i].finalcolour = c;
        break;
      }
    }
  }
  clock_t end = clock();
  double time = (double)(end - start) / CLOCKS_PER_SEC;
  cout << time << "\n";
```

```
    }
  }
  return 0;
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include <fstream>
#include <vector>
#include <time.h>
#include <sstream>
#include <string>

using namespace std;
class vertex {
public:
    int* colours;
    vector<int> confilict;
    int finalcolour;
};

int main(int argc, char** argv)
{
    int n = 0;
    const char* temp[12] = { "16.txt", "32.txt", "64.txt", "128.txt", "256.txt", "512.txt", "1024.txt",
"2048.txt", "4096.txt", "8192.txt", "16384.txt", "32768.txt"};
    for (int x = 0; x < 12; x++) {
        cout << "starting " << temp[x] << "\n";
```

```
ifstream input(temp[x], ifstream::in);

input >> n;
// int ln = ((log2(n)*3)+1);
int ln = 0;
int deg = -1;
input >> deg;
srand((int)time(0));
char edges[n][n];
for (int i = 0; i < n; i++) {
    for (int c = 0; c < n; c++) {
        input >> edges[i][c];
        edges[i][c] -= '0';
        edges[c][i] = edges[i][c];
    }
}

input.close();

for (int runs = 0; runs < 25; runs++) {
    vector<vertex> vertarr;

    vector<int> sets[deg + 1];

    //initalise vertexs
    for (int i = 0; i < n; i++) {
        vertex v;
    vertarr.push_back(v);
    vertarr[i].colours = new int[ln];
    vertarr[i].finalcolour = -1;
}

clock_t start = clock();

//assign random colours
for (int i = 0; i < n; i++) {
    for (int c = 0; c < ln; c++) {
        int colour = rand() % (deg + 1);
        sets[colour].push_back(i);
        vertarr[i].colours[c] = colour;
    }
}

//fill confilct sets
for (int i = 0; i < (deg + 1); i++) {
    for (int c = 0; c < sets[i].size(); c++) {
        for (int t = c; t < sets[i].size(); t++) {
            if (edges[sets[i].at(c)][sets[i].at(t)]) {
                vertarr[sets[i].at(c)].confilict.push_back(sets[i].at(t));
                vertarr[sets[i].at(t)].confilict.push_back(sets[i].at(c));
            }
        }
    }
}
```

```
int x = 0;
for (int cur = 0; cur < ln; cur++) { //for each colour
   for (int i = 0; i < vertarr.size(); i++) { //for each vertex
      for (int c = 0; c < vertarr[i].confilict.size() && x == 0; c++) { //for each nebour
         if (vertarr[vertarr[i].confilict.at(c)].finalcolour == vertarr[i].colours[cur]) {
            x = 1;
         }
      }
      if (vertarr[i].finalcolour == -1 && x == 0) {
         vertarr[i].finalcolour = vertarr[i].colours[cur];
      }
      x = 0;
   }
}

clock_t end = clock();
double time = (double)(end - start) / CLOCKS_PER_SEC;

int bads = 0;
for (int i = 0; i < n; i++) {
   if (vertarr[i].finalcolour == -1) {
      bads++;
      int col[deg + 1];

      for (int cl = 0; cl < deg + 1; cl++) {
         col[cl] = -1;
      }
```

```
                    for (int c = 0; c < n; c++) {
                        if (edges[i][c] && vertarr[c].finalcolour != -1) {
                            col[vertarr[c].finalcolour] = 1;
                        }
                    }

                    for (int c = 0; c < deg + 1; c++) {
                        if (col[c] == -1) {
                            vertarr[i].finalcolour = c;
                            break;
                        }
                    }
                }

                //cout << vertarr[i].finalcolour << " ";
            }

            clock_t end2 = clock();
            double time2 = (double)(end2 - start) / CLOCKS_PER_SEC;

            cout << time << "," << time2 << "," << bads << "\n";
        }
    }
    return 0;
}
```