Design and Analysis of Algorithms for Packing Coloring

theroyakash

Indian Institute of Technology Madras

January 3, 2024

Introduction to Packing Coloring

We start with a few definitions. First we define what is graph coloring.

Introduction to Packing Coloring

We start with a few definitions. First we define what is graph coloring.
Let $G = (V, E)$ be an undirected graph.

Introduction to Packing Coloring

We start with a few definitions. First we define what is graph coloring.
Let $G = (V, E)$ be an undirected graph.

Graph Coloring

Definition 1.1: Vertex Coloring

A vertex k coloring of G is a map $f : V \to \{1, \ldots, k\}$. A coloring f is said to be proper, if for every edge $(u, v) \in E$, $f(u) \neq f(v)$. The chromatic number of a graph is the minimum value of k such that G has a proper k colouring.

Graph Packing Coloring

- Given the centrality of graph colouring to graph theory and computer science, there have been several variants of colouring problems on graphs with some additional conditions imposed.

Graph Packing Coloring

- Given the centrality of graph colouring to graph theory and computer science, there have been several variants of colouring problems on graphs with some additional conditions imposed.
- One such variant is the packing coloring problem.

Graph Packing Coloring

- Given the centrality of graph colouring to graph theory and computer science, there have been several variants of colouring problems on graphs with some additional conditions imposed.
- One such variant is the packing coloring problem.
- List coloring, path coloring, repetition free colouring are some of the other prominent examples.

Graph Packing Coloring

We begin with a definition of the graph packing-coloring problem.

Graph Packing Coloring

Definition 1.2: S-Packing Coloring and Packing Coloring

Suppose $S = (a_i)_{i \in [1 \to \infty)}$ is a increasing sequence of integers, then S packing coloring of the graph is partition on the vertex set $V(G)$ into sets $V_1, V_2, V_3 \dots$ such that for every pair $(x, y) \in V_k$ is at a distance more than a_k. If $a_i = i$ for every $i \in [1 \to \infty)$, then we call the problem packing coloring.

Graph Packing Coloring

Definition 1.3: S-Packing Chromatic Number

If there exists an integer k such that $V(G) = V_1, V_2, V_3 \ldots V_k$, each V_i is a vertex-partition, then this partition is called S-packing, k coloring, and minimum of such k is the S-Packing Chromatic Number.

Difference between Packing Coloring and Graph Coloring

- There are several approximation algorithms for graph coloring, but there aren't any approximation algorithms for packing coloring.

Difference between Packing Coloring and Graph Coloring

- There are several approximation algorithms for graph coloring, but there aren't any approximation algorithms for packing coloring.
- Any graph is three-colorable is a NP-complete problem. So there are reasons to develop an approximation scheme for graph coloring.

Difference between Packing Coloring and Graph Coloring

- There are several approximation algorithms for graph coloring, but there aren't any approximation algorithms for packing coloring.
- Any graph is three-colorable is a NP-complete problem. So there are reasons to develop an approximation scheme for graph coloring.
- We also know from early on that the decision version of the packing coloring is a NP complete problem.

Difference between Packing Coloring and Graph Coloring

- There are several approximation algorithms for graph coloring, but there aren't any approximation algorithms for packing coloring.
- Any graph is three-colorable is a NP-complete problem. So there are reasons to develop an approximation scheme for graph coloring.
- We also know from early on that the decision version of the packing coloring is a NP complete problem.
- The decision version in the form of *A graph G and a positive integer K, does G have a packing-K coloring*, is NP-complete for $k = 4$ even when restricted to *planner* graphs.

Packing Coloring on Trees

Packing coloring is also NP*-Hard for the case of trees (which are acyclic undirected unweighted graphs).*

Packing Coloring on Trees

Packing coloring is also NP-Hard for the case of trees (which are acyclic undirected unweighted graphs).
It is one thing to compute the decision version of the packing coloring problem for which we don't have any efficient algorithm. It is equally or more difficult to have a fast algorithm for getting hold of an valid packing coloring assignment.

Packing Coloring on Trees

In this presentation I'll show an algorithm that gets us a valid packing coloring assignment in polynomial time.

Packing Coloring on Trees

In this presentation I'll show an algorithm that gets us a valid packing coloring assignment in polynomial time. However the algorithm do not get us the minimum number of colors that is the packing chromatic number. It is ***an approximation*** *of the actual packing chromatic number.*

Simple Algorithm for Packing Coloring

We are to find a valid assignment of packing coloring to the vertices of the graph.

Simple Algorithm for Packing Coloring

We are to find a valid assignment of packing coloring to the vertices of the graph. Most straight forward algorithm we can think of is simply assign some color, and backtrack and re-color in case of conflicts.

Simple Greedy Heuristic for Packing Coloring

- *We'll start our algorithm to work specifically on trees first because it is easier to analyze and then we'll extend it to general graphs.*

Simple Greedy Heuristic for Packing Coloring

- *We'll start our algorithm to work specifically on trees first because it is easier to analyze and then we'll extend it to general graphs.*
- *In trees every odd layer we can color with a single color* 1 *as every odd layer nodes are at distance more than* 1.

Simple Greedy Heuristic for Packing Coloring

- *We'll start our algorithm to work specifically on trees first because it is easier to analyze and then we'll extend it to general graphs.*
- *In trees every odd layer we can color with a single color* 1 *as every odd layer nodes are at distance more than* 1*.*
- *Hence number of Node remains to be colored is significantly less than the total nodes* (n)*. For example a complete* 3*-ary tree we can color* 75% *of the nodes with color* 1*.*

Simple Greedy Heuristic for Packing Coloring

- *We'll start our algorithm to work specifically on trees first because it is easier to analyze and then we'll extend it to general graphs.*
- *In trees every odd layer we can color with a single color* 1 *as every odd layer nodes are at distance more than* 1.
- *Hence number of Node remains to be colored is significantly less than the total nodes* (n). *For example a complete* 3*-ary tree we can color* 75% *of the nodes with color* 1.
- *We first see how algorithm working on a complete trees. Then we'll look into some of the optimizations we can do to improve the performance of the algorithm.*

Simple Greedy Heuristic for Packing Coloring

Algorithm 1: BASIC GREEDY ALGORITHM FOR ANY TREE

Input: Tree T

Compute Level order traversal of Tree T;
Color Every Odd layer nodes with **COLOR(1)**;
`level` $\leftarrow d - 1$ (d is the last level);
while *level* ≥ 0 **do**
  `maximum_permissible_color` $= n$;
  `current_color` $= 2$;
  foreach *Node in this level* **do**
    while *current_color* $<$ *maximum_permissible_color* **do**
      Travel to every node within distance (`int`) `current_color` and check if there is any node colored with color `current_color`;

Simple Greedy Heuristic for Packing Coloring

Algorithm 2: BASIC GREEDY ALGORITHM FOR ANY TREE

Travel to every node within distance (`int`) `current_color` and check if there is any node colored with color `current_color`;

if *None of the node is colored with color* `current_color` **then**

  Color this node with color `current_color`;

  break from the loop, go to next node in level;

else

  `current_color` ← `current_color` +1;

level ← level −1;

Output: Output this coloring assignment.

Analysis of the Basic Algorithm

During the analysis we find that there are optimizations we can do to improve the run-time of our algorithm.

Complexity Analysis

Our algorithm for each node $i \in (1, n)$ in the worst case visits all the n node to find a color (from $1 \rightarrow n$). Hence worst case time complexity is $O(n^3)$.

Optimizations 1

We observe one simple fact, that for any complete tree, the maximum number of nodes at any level is present at the last level ($= x^d$, x is the number of children and d is the depth of the last level starting root from 0).

Optimizations 1

We are coloring every odd layer with color 1.

Optimizations 1

We are coloring every odd layer with color 1.
Instead of that if we color the last level and then every alternate level with color 1 we'll color much more nodes with color 1 and reduce the total number of colors used. Here is a simple example how this optimization saved thousands of colors.

Optimizations 1

Nodes	Layers	Maximum Colors used	Runtime
265720	12	20633	52m 32s 280ms
265720	12	6890	4m 17s 31ms

Optimizations 1

Nodes	Layers	Maximum Colors used	Runtime
265720	12	20633	52m 32s 280ms
265720	12	6890	4m 17s 31ms

This one simple optimization reduces the runtime by 92%

Optimizations 2

Suppose we are at the moment trying to color node u. Our algorithm for each color $i \in (1, n)$ goes to distance i from the node u and checks if that color exists already or not in all the nodes sitting within distance i from node u?

Optimizations 2

Suppose we are at the moment trying to color node u. Our algorithm for each color $i \in (1, n)$ goes to distance i from the node u and checks if that color exists already or not in all the nodes sitting within distance i from node u?

Lets see this step of the basic algorithm with an example.

Optimizations 2

- Suppose we are currently looking to color some node u with color d.

Optimizations 2

- Suppose we are currently looking to color some node u with color d.
- We went to d distance from node u to find all the colors we find. Suppose we find color $d, d+1, \ldots, d+k$ are present in some of the nodes.

Optimizations 2

- Suppose we are currently looking to color some node u with color d.
- We went to d distance from node u to find all the colors we find. Suppose we find color $d, d+1, \ldots, d+k$ are present in some of the nodes.
- So we should not check this again for node u with color $d+(1 \rightarrow k)$.

Optimizations 2

- Suppose we are currently looking to color some node u with color d.
- We went to d distance from node u to find all the colors we find. Suppose we find color $d, d+1, \ldots, d+k$ are present in some of the nodes.
- So we should not check this again for node u with color $d + (1 \rightarrow k)$.
- Hence we implement this modification to improve the runtime.

Optimizations 2

We define a subroutine called `Check(u, d)`*. This subroutine returns a set of colors present within distance d for any node u.*

Optimizations 2

We define a subroutine called ***Check(u, d)****. This subroutine returns a set of colors present within distance d for any node u.*

Algorithm 4: `Check(Node u, Color d)`

$\mathcal{C} \leftarrow \phi$;
Visit all nodes within distance d from node u and collect all the colors into $\mathcal{C}$;
return Set $\mathcal{C}$;

Optimizations 2

We define a subroutine called ***`Check(u, d)`****. This subroutine returns a set of colors present within distance d for any node u.*

Algorithm 5: `Check(Node u, Color d)`

$\mathcal{C} \leftarrow \phi$;
Visit all nodes within distance d from node u and collect all the colors into $\mathcal{C}$;
return Set $\mathcal{C}$;

We can call this subroutine from the main coloring BFS call (we are coloring left to right, level by level). We start with the color 2 and then we follow the following coloring strategy.

New coloring strategy

Algorithm 6: `Updated Main Coloring Scheme`

for *Each node from last uncolored level, left to right* **do**
- $d_{\text{prev}} \leftarrow \phi$;
- **for** *Each Color i from* $2 \rightarrow n$ **do**
  - **if** *Color* $i \in d_{prev}$ **then**
    - It is not possible to color this node with color i because we found color i at distance less than i in d_{prev};
    - Continue with color $i + 1$;
  - **else**

New coloring strategy

Algorithm 7: `Updated Main Coloring Scheme`

for *Each node from last uncolored level, left to right* **do**
    $d_{\text{prev}} \leftarrow \phi$;
    for *Each Color* i *from* $2 \rightarrow n$ **do**
        if *Color* $i \in d_{prev}$ **then**
        else
            $d_{\text{new}} =$ `Check(node, i,` d_{prev}`);`
            if $i \notin d_{new}$ **then**
                Color this node with color i;
                Break from this loop and start coloring next uncolored node;
            else
                $d_{\text{prev}} = d_{\text{new}}$

Optimization 3

- This optimization comes from the observations of the structure of the complete trees.

Optimization 3

- This optimization comes from the observations of the structure of the complete trees.
- Complete x-ary trees has a depth of $\log_x n$ with n many nodes in them.

Optimization 3

- This optimization comes from the observations of the structure of the complete trees.
- Complete x-ary trees has a depth of $\log_x n$ with n many nodes in them.
- With the following optimization our algorithm time complexity will reduce from $O(n^3)$ down to $O(nd^2)$ for x-ary trees with d diameter. This is a significant complexity improvement.

Optimization 3

- Suppose j is a color that has been used in the tree for the first time (during our run of the algorithm).

Optimization 3

- Suppose j is a color that has been used in the tree for the first time (during our run of the algorithm).
- If $j >$ the longest path in the tree, then color j can never be used again.

Optimization 3

- Suppose j is a color that has been used in the tree for the first time (during our run of the algorithm).
- If $j >$ the longest path in the tree, then color j can never be used again.
- Any color after j that is $j + 1$ and so on will also not be possible to reuse.

Optimization 3

- Suppose j is a color that has been used in the tree for the first time (during our run of the algorithm).
- If $j >$ the longest path in the tree, then color j can never be used again.
- Any color after j that is $j + 1$ and so on will also not be possible to reuse.
- So there is a upper bound on the number of color that are reusable. This depends on the longest path on the tree.

Sorting + Searching

Before introducing other data structure in the STL library, I'll show you some algorithms and iterator access on vector which is used often.

- Iterators
- Sorting
- Searching

Iterators

- Similar to pointers in C, C++ has Iterators

Iterators

- Similar to pointers in C, C++ has Iterators
- `vector<int>::iterator it = v.begin();` returns a *Pointer* to the first element of vector,

Iterators

- Similar to pointers in C, C++ has Iterators
- `vector<int>::iterator it = v.begin();` returns a *Pointer* to the first element of vector,
- Or you can use `auto it = v.begin();`

Iterators

- Similar to pointers in C, C++ has Iterators
- `vector<int>::iterator it = v.begin();` returns a *Pointer* to the first element of vector,
- Or you can use `auto it = v.begin();`
- For example if you have an array $v = [10, 12, 13, 14]$ then `*it` would return 10,

Iterators

- Similar to pointers in C, C++ has Iterators
- `vector<int>::iterator it = v.begin();` returns a *Pointer* to the first element of vector,
- Or you can use `auto it = v.begin();`
- For example if you have an array $v = [10, 12, 13, 14]$ then `*it` would return 10,
- Similar to pointer you can increase and decrease them, `it++;` and then `*it` would return 12.

Last Iterator

- `vector<int>::iterator it = v.end();` points to a non-existent element sits after the last element.

Last Iterator

- `vector<int>::iterator it = v.end();` points to a non-existent element sits after the last element.
- Hence `*it` would dereference nothing when `it = v.end();`.

Sorting

Sorting and searching is the most common thing you do on a vector.

```
#include <algorithm>

int main () {
    vector<int> v = {4,3,2,1};
    std::sort(v.begin(), v.end());
    for (auto i : v) { cout << i << " "; }
}
```

Custom Sorting

What to do when you have a vector of custom data structure?

Custom Sorting

What to do when you have a vector of custom data structure?

For that we need to design custom comparators.

Custom Comparators Showcase

```
int main() {
    vector<pair<int, int>> v = {{1,2}, {-3,4}, {-12, 12}};
    sort(v.begin(), v.end(), [](const auto &a, const auto &b) {
        return a.first < b.first;
    });
}
```

Custom Comparators Showcase II

Using this you can define your own rule for sorting, for example following shows how to sort in decreasing order.

Custom Comparators Showcase II

Using this you can define your own rule for sorting, for example following shows how to sort in decreasing order.

```
int main() {
    vector<int> v = {1,2,3,4};
    std::sort(v.begin(), v.end(), [](const auto &a, const auto &b) {
        return a > b;
    });
}
```

Custom Comparators Showcase III

Following is an example of sorting a custom class of data.

Custom Comparators Showcase III

Following is an example of sorting a custom class of data.

```
class Job {
public:
    int timestamp; int jobID; vector<int> requests;
};

int main() {
    vector<Job> jobs;
    std::sort(jobs.begin(), jobs.end(), [](const auto &a, const auto &b) {
        return a.timestamp < b.timestamp; // sort according to timestamp
    });
}
```

Note

This comparator should return true for argument (a, b) if and only if a sits left of b in the sorted array.

`std::lower_bound` and `std::upper_bound`

- We need a non-decreasingly sorted container,
- We want to find out position of the smallest number just $>$ (greater) a given number or position of the smallest number $\geq$ (greater than or equal to) a given number

`std::lower_bound` and `std::upper_bound`

- `std::lower_bound` returns iterator to first element in the given range which is equal or greater than the value.
- `std::upper_bound` returns iterator to first element in the given range which is greater than the value.

List container

- List is a doubly linked list, this includes functions such as `push_front`, `push_back`, `insert`, `erase`, etc.

List container

- List is a doubly linked list, this includes functions such as `push_front`, `push_back`, `insert`, `erase`, etc.
- However I'd say not to use `std::list` as a data structure because difficult to manipulate.

Few Example

Following is the usage of `front()`, `back()` on a list. This runs in $O(1)$ time.

Few Example

Following is the usage of `front()`, `back()` on a list. This runs in $O(1)$ time.

```
#include <list>
#include <iostream>

int main() {
    std::list<char> letters {'d', 'm', 'g', 'w', 't', 'f'};

    if (!letters.empty()) {
        std::cout << "The first character is '" << letters.front() << "'.\n"
    ;
        std::cout << "The last character is '" << letters.back() << "'.\n";
    }
}
```

Singly Linked List

- Similarly there is singly linked list called `std::forward_list`.

Singly Linked List

- Similarly there is singly linked list called `std::forward_list`.
- It is not recommended to use these list data structures as you have limited control on the pointers.

Example Usage

Finding Middle of linked list

```
ListNode *middleNode(ListNode *head) {

    if (!head->next) {
        return head;
    }

    ListNode *slowPointer = head;
    ListNode *fastPointer = head;

    while (fastPointer != NULL && fastPointer->next != NULL) {
        slowPointer = slowPointer->next;
        fastPointer = fastPointer->next->next;
    }

    return slowPointer;
}
```

Example Usage on STL List

Finding Middle of STL Forward List

```
template <class T>
T findMiddleElement(forward_list<T> *list) {

    // Using 2 pointer approach
    auto slowPointer = list->begin();
    auto fastPointer = list->begin();
```

Example Usage on STL List

```
    // Update the slowPointer slowly and fastPointer quickly

    while (fastPointer != list->end() &&
           std::next(fastPointer, 1) != list->end()) {

        std::advance(slowPointer, 1);
        std::advance(fastPointer, 2);
    }

    return *slowPointer;

}
```

Map and Set

- Associative containers (Map and Set) are not sequential; they use keys to access data.

Map and Set

- Associative containers (Map and Set) are not sequential; they use keys to access data.
- Both Maps and Sets are internally implemented using trees,

Map and Set

- Associative containers (Map and Set) are not sequential; they use keys to access data.
- Both Maps and Sets are internally implemented using trees,
- Allows for efficient searching, insertion and deletion

Set Example

- Stores unique values

Set Example

- Stores unique values
- Interface functions: `insert`, `erase`, `clear`, `find`, `upper_bound`, `lower_bound`.

Example Usage

```
#include <set>
using namespace std;

int main () {
    set<int> marks;
    marks.insert(10); marks.insert(20); marks.insert(30);
    marks.insert(100); marks.insert(100);

    for (auto elem : marks) { cout << elem << " "; }

    return 0;
}
```

Example Usage

Storing from Larger to Smaller

```
#include <set>
using namespace std;

int main () {
    set<int, greater<int>> marks;
    marks.insert(10); marks.insert(20); marks.insert(30);
    marks.insert(100); marks.insert(100);

    for (auto elem : marks) { cout << elem << " "; }

    return 0;
}
```

std::multiset

Multiset works same as a set but allows duplicate values.

```
#include <set>

int main () {
    multiset<int, greater<int>> marks;
    marks.insert(100); marks.insert(100); marks.insert(100);
    marks.insert(20); marks.insert(30);

    // 100 100 100 30 20
    for(auto e : marks) cout << e << " ";

    return 0;
}
```

`std::map`

To store Key-Value pairs you have two options `std::map` and `std::unordered_map`.

`std::map`

In `std::map` when you access the elements you get the values in sorted order.

std::map

- Each key is unique

std::map

- Each key is unique
- Internally implemented using `Red-Black Trees`

`std::map`

- Each key is unique
- Internally implemented using `Red-Black Trees`
- Interface contains functions such as `find`, `count`, `clear`, `erase`, etc,
- Also supports array-like indexing with keys.

std::unordered_map

- Works same as the map, but has better time complexity when access, find, and erase is called.

std::unordered_map

- Works same as the map, but has better time complexity when access, find, and erase is called.
- Each of the operations on `std::map` is $O(\lg n)$ but on `std::unordered_map` each access, find, erase is amortized $O(1)$.

Amortization

- With `C++11` we got hash set and hash map in the form of `std::unordered_set` and `std::unordered_map`

Amortization

- With `C++11` we got hash set and hash map in the form of `std::unordered_set` and `std::unordered_map`
- By amortized $O(1)$ insertion time we mean that each item only runs into $O(1)$ collisions on average,

Amortization

- With `C++11` we got hash set and hash map in the form of `std::unordered_set` and `std::unordered_map`
- By amortized $O(1)$ insertion time we mean that each item only runs into $O(1)$ collisions on average, This means there exists a sequence of elements for which `std::unordered_map` has collisions in every insertion.

Other hash functions

The **builtin** hash function for C++ is not optimal. There exists far better hash functions one such example is `SplitMix64`.

SplitMix64

Example Usage

```
struct SplitMix64 {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().
    time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
```

SplitMix64

Redefinition With `SplitMix64`

Now you define the `unordered_map` as `unordered_map<int, int, SplitMix64> unmap;`