Recitation 3 Guide: Greedy Algorithms

Fall 2022

This week we will be learning about greedy algorithms. A greedy algorithm is an algorithm which makes the locally optimal choice at each step, without regard for global optimality structure. These algorithms sometimes produce an optimal solution; frequently they do not produce optimal solutions. We will go over the conditions for when they work, and examine some scenarios where greedy algorithms do not work.

1 Toy Problems

To study greedy algorithms, we will examine some very specific problems.

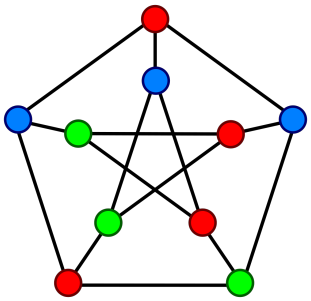
Problem 1.1. (Graph Coloring) Given a graph $G = (V, E)$, we want to color the vertices of G with the fewest possible colors, subject to the condition that no two adjacent vertices may receive the same color.

Formally, a coloring of $G = (V, E)$ is a function

$$f : V \longrightarrow C$$

where C is a set of colors.

Example 1.2. The Petersen Graph, below, can be coloured with the set $C = \{red, blue, green\}$ so that no two adjacent vertices get the same color.

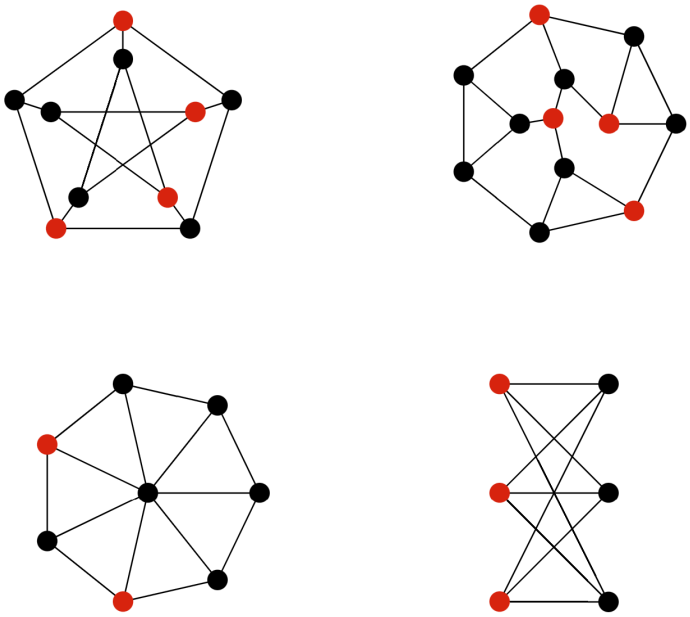


Definition 1.3. We say that a graph $G = (V, E)$ is **correctly-colored** if no two adjacent nodes receive the same color.

Definition 1.4. If a graph G is correctly-colorable with k colors, we say that G is "k-partite", or "splits into k parts". For example, the Petersen Graph above is tripartite (3-partite).

Problem 1.5. (Maximum Independent Set) Given a graph $G = (V, E)$, find a subset $S \subseteq V$ such that none of the vertices in S are adjacent to each other. Maximize the number of vertices in S.

Example 1.6. Some independent sets in various graphs, shown in red. Notice how none of the red vertices are touching each other with an edge. You may notice that the independent set in the top left graph is exactly the red color class from the 3-coloring on the previous page.



Remark 1.7. The **Graph Coloring** problem is the problem of partitioning the set of vertices into independent sets. In a correctly-colored graph, each color class is an independent set. Thus, we want to partition the graph into as few independent sets as possible.

When viewed like this, the **Maximum Indepedent Set** problem is the problem of legally coloring as many of the vertices as you possibly can, but you only get one color, $\{red\}$.

1.1 Applications of Graph Coloring

1. **Exam Scheduling:** CU has to come up with exam schedules every semester, and they want to minimize schedule conflict. This is actually a graph coloring problem in disguise. We will make a graph whose vertices are the classes offered at CU. There will be an edge between two vertices if the corresponding classes share any students. Thus, we want to come up with a partition of the classes into independent sets (i.e. color classes). Each color class will correspond to an exam time, and so we don't want any pair of classes which share a student to have their exams at the same time.

2. **Assigning Radio Frequencies:** The problem of assigning radio frequencies to radio stations so that no two stations which broadcast in the same area get the same frequency. a color, a frequency range

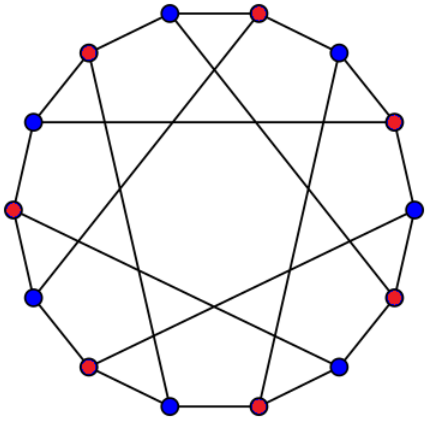
3. **Compiler Design:** When designing a compiler, one has to allocate a small number of CPU registers to process a large number of program variables. These program variables can interfere with each other, and so edges are added between a pair of program variables if they are simultaneously live at at least one program point. Allocating variables to registers so that they don't interfere is a graph coloring problem.

4. **Sudoku:** It turns out Sudoku is secretly a graph coloring problem. Specifically, it is a 9-coloring problem. What are the nodes and edges of this graph?

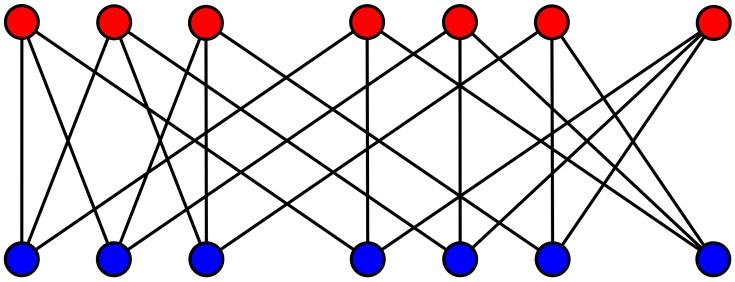
1.2 Bipartite Graphs

Of special interest is graphs which can be colored with 2 colors, i.e. bipartite graphs. These types of graphs tend to have very simple structure.

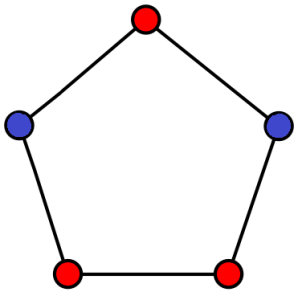
Example 1.8. The following graph is bipartite



Bipartite graphs are often drawn with all the vertices of one color on one side, and all the vertices of the other color on the other side. This makes it obvious that the graph is partitioned into two independent sets. Here is the same graph as above, partitioned according to color.



Remark 1.9. We can observe that any odd-length cycle cannot be bipartite, because we cannot color it with two colors. If we color the top vertex red, then the middle two MUST be blue, and then the bottom two MUST be red, but the bottom two being red is not a correct-coloring.

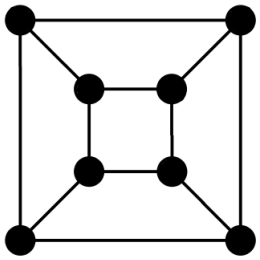


We have established that if a graph has an odd-length cycle embedded in it, it cannot be bipartite. Remarkably enough, it turns out that the ONLY obstacle to graphs being bipartite is odd-length cycles. That is,

Theorem 1.10. *A graph G is bipartite (2-colorable) if and only if it contains no odd-length cycle.*

This theorem is an example of something called a **good characterization**, which links the existence of one thing (a 2-coloring) to the nonexistence of something else (the embedding of an odd cycle). That is, a graph can be 2-coloured if and only if it contains no odd cycles. Good characterizations are useful, **as we will see in the third exercise** since proving the nonexistence of one thing amounts to exhibiting existence of the other thing.

1.3 The Cube



This graph is known as "The Cube". We will see it many times on the next page. It is bipartite.

2 Maximal Solutions vs. Maximum Solutions

The distinction between maximAL and maximUM solutions (and, conversely minimal and minimum solutions) is important in determining whether a greedy algorithm will find the best possible solution to a problem.

Definition 2.1. A solution is **maximal** if it cannot be expanded to a larger solution.

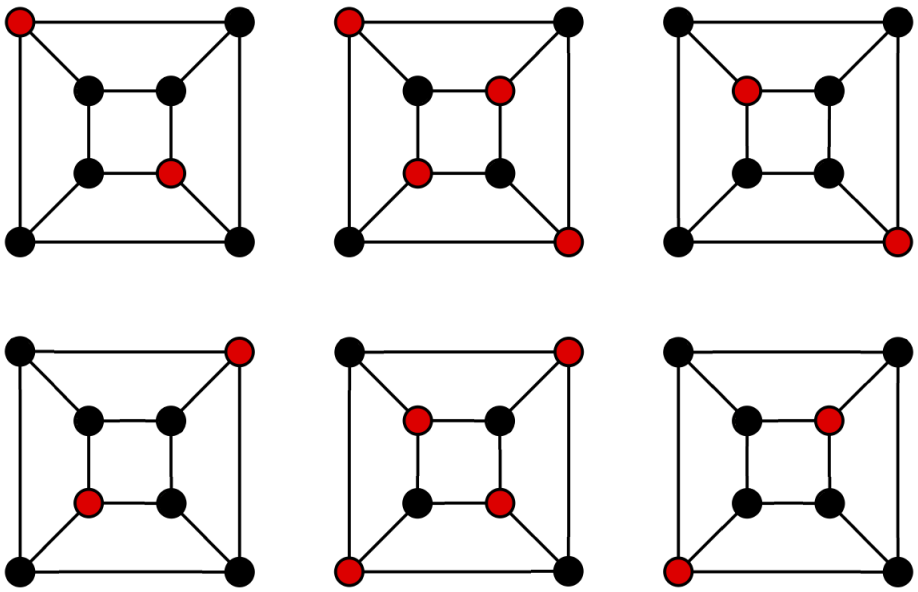
Likewise, a solution is **minimal** if it cannot be reduced to a smaller solution.

Definition 2.2. A solution is **maximum** if it has the greatest value among the set of all possible solutions (i.e. a larger solution does not exist).

Likewise, a solution is **minimum** if it has the smallest value among the set of all possible solutions (i.e. a smaller solution does not exist).

Greedy algorithms will find maximal / minimal solutions. This does not necessarily mean the solutions will be maximum / minimum.

Example 2.3. The cube graph has several maximal independent sets, shown in red below. All of them are maximal, in that we cannot add any vertices to them. Only two of them (the ones in the middle, of size 4) are maximum. The others (of size 2) are maximal but not maximum.



Even the independent sets of size 2 cannot be expanded upon. There is no vertex which can be added to any of these independent sets which does not violate the definition of an independent set.

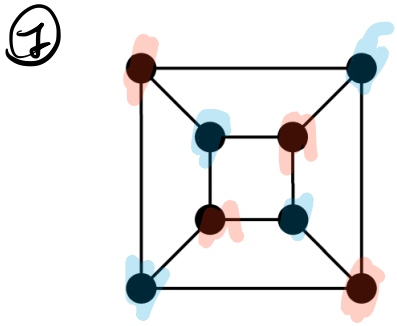
If you can prove that all maximal solutions are maximum, then you have proven that a greedy algorithm will find the maximum solution.

3 Exercises

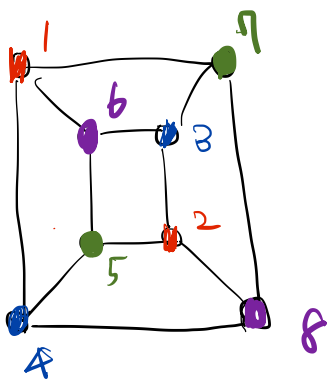
Exercise 3.1. Consider a greedy algorithm for Graph Coloring

```
Take list of vertices V in order: V[1], V[2], ...
Take list of colors C in order C[1], C[2], ...

For i = 1 to |V|
Color V[i] with the first available legal color.

\\(That is, check if it is legal to color V[i] with C[1]. If not, try C[2].
If not, try C[3]. And so on...)
```

Consider the cube graph:



1. **This graph is bipartite. Show that it is bipartite by correctly-coloring its vertices with only two colors.**
2. **Find an ordering of the vertices of the cube such that the greedy algorithm above gives it four colors. Think about the independent sets which are maximal but not maximum.**
3. **Discuss in what sense the greedy 4-coloring you found is minimal, but not minimum. (Open-ended)**



The resulting 4-coloring is minimal in the sense that no color class can be eliminated and re-colored with the remaining three colors.

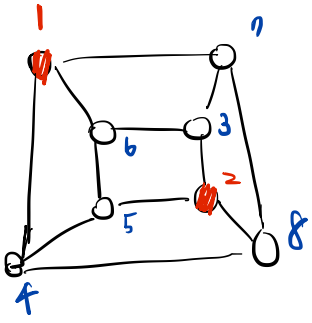
Exercise 3.2. Consider a greedy algorithm for Independent Set.

```
Input: G = (V,E)

S <-- []
For i = 1 to |V|
    b <-- TRUE
    For j = 1 to |S|
        if V[i] is adjacent to S[j]
            b <-- FALSE
        else
            do nothing

    if b == TRUE
        S.append(V[i])

Return S
```

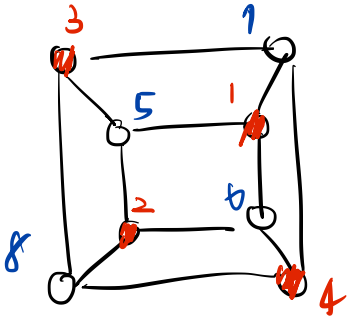
This algorithm runs through the list of vertices in order, checking whether each vertex is adjacent to any of the vertices it has already selected to be in the independent set S. If it finds a vertex $V[i]$ which is not adjacent to anything in S, it adds $V[i]$ to S.

1. **Come up with a graph G (not the cube graph from above) and an ordering of the vertices V such that the above algorithm finds a maximal independent set which is not maximum.**
2. **On the same graph, order the vertices in a different way so that the algorithm finds a maximum independent set.**
3. **How many of the possible orderings of V do you expect to result in finding a maximum independent set? (Open-ended)**

(1)



(2)



(3)

?

Exercise 3.3. We will look at the problem of deciding whether a given graph can be 2-colored.

1. **Design a greedy algorithm which, given a graph $G = (V, E)$, determines whether G is bipartite (i.e. two-colorable). Make use of BFS and/or DFS.** Hint: Your algorithm should be looking for one of two things:
   - A 2-coloring (meaning that G is 2-colorable)
   - An odd-length cycle (meaning that G is not 2-colorable)
2. **Prove that your algorithm is correct.**
3. **Discuss why your methodology cannot be extended to testing whether a graph is 3-colorable. (Open-ended)**

Remark 3.4. There is a very interesting gap between 2-colorability and 3-colorability. Deciding whether a given graph is 2-colorable can be done greedily and efficiently with a BFS. Deciding whether a graph is 3-colorable is NP-complete, which means that we do not currently have an efficient algorithm for it, and we expect that one does not exist.

① Apply BFS starting from a certain vertex, Mark the visited vertices with an avaible color, which could be blue or red. If the node has no neighbors, we color it red. If the node has neighbors colored in red, then we color it blue. If the node has neighbors colored in blue and red, then return False. If we end up coloring all nodes, then return True.

②
- We can color a node in red iff it does not have neighbors.
- We can color a node in blue iff it has only red or uncolored neighbors
- we can't color a node, iff it has red and blue neighbors

BFS always visit their neighbors first before visiting others. Neighbors are also the only groups, which we can deside which color to assign for.