September 28, 2023

1 Graph Coloring Problem

The Cricket Tournament problem can be transformed into a graph coloring problem. In the constructed graph, the nodes of the graph correspond to the games to be played, and an **edge between two nodes represents that the games cannot be scheduled together**. Here, each color will represent the set of games that can be scheduled together in a two-day span.
For example, n=3, The three teams are the Delhi Capitals (DC), Kolkata Knight Riders (KKR), and Mumbai Indians (MI). Then, the following games have to be scheduled as stated below:
DC vs. KKR, DC vs. KKR, DC vs. MI, KKR vs. DC, KKR vs. MI, MI vs. DC, and MI vs. KKR.

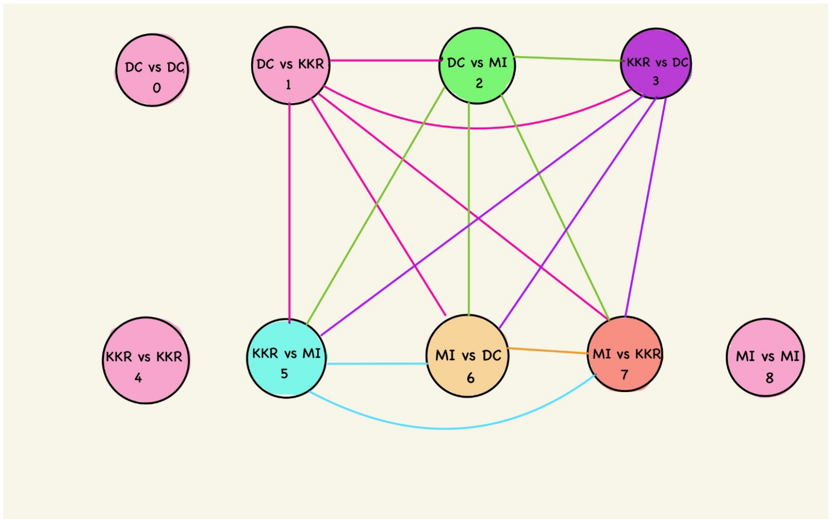

Figure 1: This graph depicts the graph coloring problem for three teams.

Figure 1 shows a graph with nodes corresponding to games to be played and links corresponding to games that cannot co-occur. The constraints can be satisfied by scheduling games only once per day, with a day gap in between, and the most efficient solution can be computed using the greedy approach to color the graphs. Here, the Greedy approach will give an optimal solution as there cannot be more than one game in any day as one game involves two teams. There has to be a one-day gap to avoid overlapping between scheduled games. Hence, the schedule is as follows:
DC vs KKR

DC vs MI

KKR vs DC

KKR vs MI

MI vs DC

MI vs KKR
Hence, in Figure 1 we need 6 colors to color the graph and we need a total of 11 days to schedule the cricket tournament

1.1 Process of going from problem to program

- We are given an imprecise or natural language description of a real-world problem to be solved. In this case, we have to schedule cricket tournaments between a set of teams.
- We formalize the cricket tournament problem so that we have a mathematical description of the problem. The solution requirements are are follows:
    1. Constraint: A team is allowed to play one game in a two-day span.
    2. Efficiency: The tournament should be completed in the minimum number of days.
- Then, we write a valid algorithm that gives us a high-level description of our problem. The algorithm is the solution process which is human understandable.

  This is an optimization problem. Out of all the feasible solutions with constraints specified in (1.), we have to satisfy the optimization condition i.e. (2.). This problem is an example of the constraint optimization problem. The most efficient solution is hard to find, so we adopt some heuristics. Given the constraints, we have to find the most efficient solution which is equivalent to the Graph coloring problem. The heuristic we adopt is a greedy heuristic.
- Pseudocode is like a flowchart or execution flow of our program, which has solution control flow and ADTs- Abstract Data Types. The ADTs we will be using to implement our solution are GRAPH and LIST. In the final program code, we implement these ADTs without using any in-built library.
- Finally, we write the final code using relevant data structures. The process of programming is depicted in Figure 2.

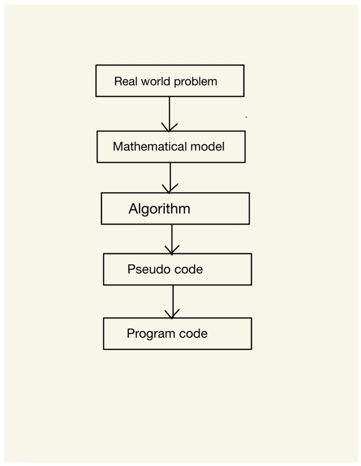

Figure 2: Process of programming

2 LIST and GRAPH Abstract Data types(ADTs)

2.1 LIST

The data structure that we have used to implement our List ADT is a linked list. The basic cell of the list consists of a *data* field and a *pointer* field which points to the next cell. To form an abstract data type from the mathematical notion of a list we define a set of operations on objects of type LIST. The operations that we have defined for our List ADT are as follows:

- **push_back(*datatype* x)**: This method adds elements to the list.
- **get(int index)**: This method returns the value present at index specified.

- **set(int position, int value)**: This method allows us to assign a value given a particular index.
- **print()**: Prints the list.
- **getSize()**: Returns the size of the list.

The Pseudo code for these operations are as follows:

1. Add Elements

```
procedure PUSH_BACK()(int val)
    $Node * newNode \leftarrow newNode(value)$
    if $head! = NULL$ then
        $head \leftarrow newNode$
    else
        $Node * current \leftarrow head$
        while $current- > next$ do
            $current \leftarrow current- > next$
        $current- > next \leftarrow newNode$
    $size \leftarrow size + 1$
```

2. Get an Element

```
procedure GET(int index)
    if $index < 0$ or $index > size$ then
        $OutofRange$
    $Node * current \leftarrow head$
    for $i \leftarrow 0$ to $index - 1$ do
        $current \leftarrow current- > next$
    $return\ current- > next$
```

3. Set to a value

```
procedure SET(int index, int value)
    if $index < 0$ or $index > size$ then
        $OutofRange$
    $Node * current \leftarrow head$
    for $i \leftarrow 0$ to $index - 1$ do
        $current \leftarrow current- > next$
    $current- > data \leftarrow value$
```

4. Print

```
procedure PRINT()
    $Node * current \leftarrow head$
    while $current$ do
$cout << current- > data$
        $current \leftarrow current- > next$
$cout << endl$
```

5. Size of list

```
procedure GETSIZE()
    $return\ size$
```

2.2 GRAPH

The graph is built using an adjacency list. An adjacency list is a data structure used that represents a graph where each node in the graph stores a list of its neighboring vertices. Graph ADT makes use of the List ADT which we have built. The operations defined on Graph ADT are as follows:

- **addEdge**: It adds an edge between two vertices. The Pseudo code for GRAPH Abstract data type is as follows:

1. Add Edge

```
procedure ADDEDGE($int\ i$, $int\ j$)
$int\ nodes$
$List < int > *l$
    $l[i].push_back(j)$
    $l[j].push_back(i)$
```