CSCI203

Week 10 – Lecture B

Crazy Eights

- We are now going to examine a problem involving playing cards.
- Given a sequence of playing cards find the longest valid subsequence of cards in which each card selected is “like” its neighbours.
- Cards are like each other if:
  - Either they are of the same value;
  - Or they are of the same suit;
  - Or at least one of the cards is an 8.
- Sub sequences must be in the same left-to-right order as the original sequence but are allowed to skip cards.

An Example

- Consider the following sequence of cards:

  5♣ 5♦ 10♦ 8♥ 7♠ 9♣ A♣ K♦ J♣

- The following are some possible subsequences

  5♦ 10♦ 8♥ 7♠

  5♦ 10♦ 8♥ A♣

  5♣ 5♦ 10♦ 8♥ 9♣ A♣ J♣

- In this example the last example is the longest possible.

How to Solve it.

- How can we solve this puzzle?
- Surprise!
  - We use Dynamic Programming.
- The first (and biggest) part of this process is how do we re-state the problem in a way that we can use DP to solve.
- We need a directed, acyclic graph of dependencies.
- Once we have that, the rest is easy.
- So, how do we turn the problem into a graph?

Cards to Graphs

- To convert the problem into a graph, we need to identify:
  - The vertices;
  - The edges;
  - The weights.
- We also need to formulate the problem in terms of minimizing (or, possibly, maximizing) some function of the weights.
  - We call this the *objective* function.
- The first of these conversion issues is easy to address:
  - What are the vertices?
    - The cards.
    - Actually, we will add one more “dummy” vertex s.

What are the Edges?

- We now define directed edges (u, v) as follows.
- Edge (u, v) exists as long as:
  - Card u lies to the left of card v in the original sequence.
  - Card u is like card v.
- Remember, any 8 is like every card.
- So is our starting card s.

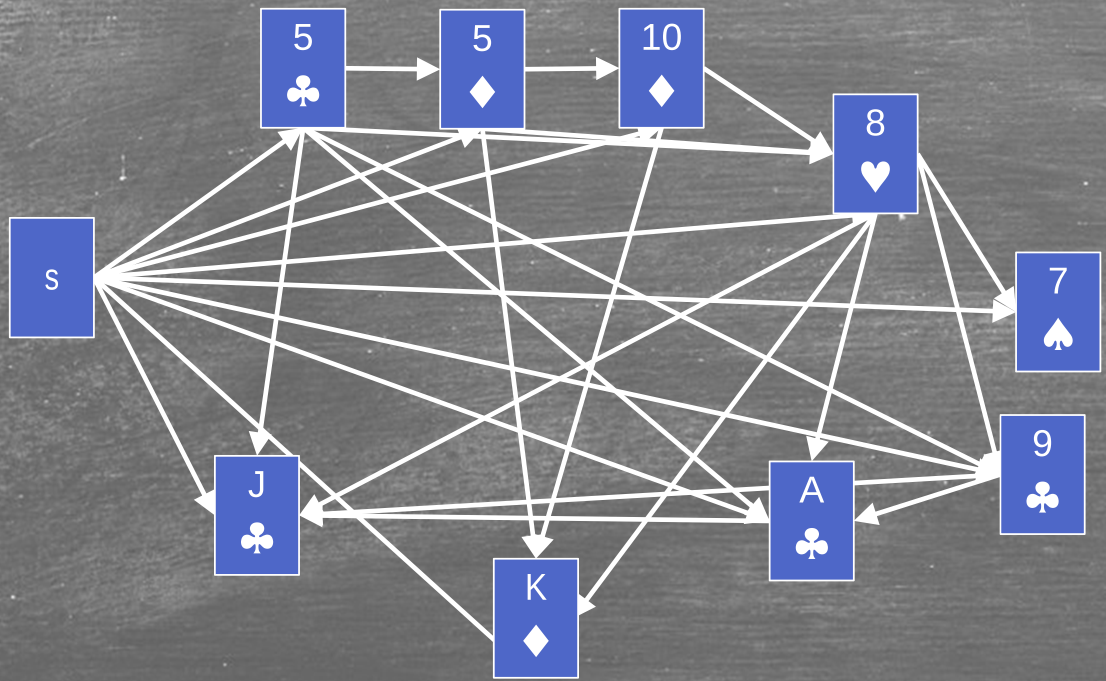
Edge Weights?

- As we are only concerned with the number of cards we do not need to differentiate one edge from another.
- Thus, we can assign a constant weight of one to each edge.
- Our example problem is converted into the following directed graph:

The Graph

5♣ 5♦ 10♦ 8♥ 7♠ 9♣ A♣ K♦ J♣

- Becomes:



The Objective Function

- For this problem we want to find the longest valid sub-sequence.
- We note that the length of a valid sub-sequence, ending at card v in the original problem, is equal to the maximum path-length, $M(s, v)$, from s to v in the graph we have defined.
- This is defined as the sum of weights (each equal to one) of the edges in the path.
- Our objective function is $M(s, v)$ and the problem becomes:
  - Find $v \in V$ such that $M(s, v)$ is maximized.

Solving the Problem

- We note that:
- If the maximum path from *s* to *v* passes through some vertex *u* then:
  - M(s, v) = M(s, u) + M(u, v).
- If *u* is the last vertex (card) before *v* then:
  - M(s, v) = M(s, u) + w(u, v) = M(s, u) + 1.
- So our problem becomes:
  - For each *v* in *V*:
    - Find M(s, v)
  - Find the maximum of all the M values.

Crazy Eights using Dynamic Programming

- The solution for vertex *v* becomes:
-
```
M: dictionary = {}
Procedure crazyDP(V{}: vertex, E{}: edge, s: vertex, v: vertex)
     for each v in V
            m=0
            if v ≠ s then
                   for each u where (u,v) ∈ E
                          if (u in M) then
                                 m = max(m, M[u] + 1)
                          else
                                 m = max(m, crazyDP(V, E, s, u) + 1)
                          fi
                   rof
            fi
            M[v] = m
     rof
End procedure crazyDP
```

Driver Program

- We now need a driver program to iterate over all the vertices.
- for each v in V
      crazyDP(V, E, s, v)
  rof
  solution = max M[v]
- We can use the standard methods to optimize the algorithm:
  - E.g. make M a max heap ordered on path length;
  - Track the preceding card in each longest sequence:

Another Approach:

- Consider the following, alternate formulation of the problem:
  - *V* is still the cards plus *s*, a dummy starting card;
  - *E* is the same as we defined before;
  - *W* is set to -1 for each edge in *E*.
- Now we have a simple shortest path problem.
- We could use Bellman-Ford to solve this.

Is This Good?

- Bellman-Ford is $O(|V| \times |E|)$
- If we have N cards.
  - $|V| \in O(N)$;
  - $|E| \in O(N^2)$
- So the overall running time is $O(N^3)$
- We can do better than this:

Better than N^3

- We note that the graph associated with the problem is acyclic:
  - All edges go from left to right.
- This means that we can find a shortest path solution in $O(|V|+|E|) = O(N+N^2) = O(N^2)$
- We just need a good order in which to consider the vertices...
  - ...we need a topological sort.
- How do we get one?

Order Zero Topological Sort

- We can get a topological sort of our problem graph in O(0) time.
- Yes—no time at all!
- How?
- We already have it.
- It is the original order of our sequence of cards!
- We can now write a non-recursive, bottom up, DP procedure that encapsulates the insights we have gained.

- Let C be the original sequence of cards:
  - $C=(c_0, c_1, c_2, c_3, \ldots, c_{N-1}, c_N)$;
  - Note: $c_0 = s$.
- We can write a bottom up algorithm as follows:

Bottom Up Crazy Eights

```
Procedure crazyUp(C(): cards)
      Length: dictionary = {}
      Length[0] = 0
      maxLength=0
      for i in 1..N
            len = 1
            for j in i-1..0
                  if C(j) is like C(i) then
                        len = max(len, Length[j] + 1)
                  fi
            rof
            Length[i] = len
            maxLength = max(maxLength, len)

      rof
      return maxLength
End procedure crazyUp
```

Notes

- We have completely eliminated the graph;
  - This is not strictly true…
  - …we have used an *implicit* representation of the graph.
- Our test:
  - **`if C(j) is like C(i)`**;
  - Is equivalent to traversing the edges on the dependency graph.
- The order of the cards:
  - **`c(0), c(1), …, c(n)`**;
  - Is the topological sort of the vertices.
- The only thing that remains of the original formulation is:
  - The dictionary, **`Length[]`**.