Universidade Nova de Lisboa
OMNIS CIVITAS CONTRA SE DIVISA NON STABIT
Faculdade de Ciências e Tecnologia

Graph Algorithms: Combinatorial Problems

Pedro Barahona
DI/FCT/UNL

Métodos Computacionais
1st Semestre 2019/2020

Combinatorial Problems: Graph Algorithms

- As discussed before, many problems, including graph problems, require choices to be done during the course of their solving, and these choices are not guaranteedly correct.
- Hence alternatives have to be searched during program execution, and these may lead to a combinatorial explosion. This is the case of two well known problems
  a. Minimum Hamiltonian tours (Traveling Salesman)
  b. Minimum number of colours
- In the first case, for a graph with **n** nodes (cities in the TSP jargon) the tour is constructed "city by city" and the choice of the next city to visit is not certain. Hence, a total of **n!** possibilities might have to be tested.
- In the second case, the colours are assigned to the nodes, one by one, and the choices might have to be changed when the colours assigned prevent the unassigned nodes to be coloured. In the worst case, assigning one of **k** colours to each of the **n** nodes of a graph, requires testing $\mathbf{k^n}$ potential combinations of colours.

Combinatorial Problems: Graph Algorithms

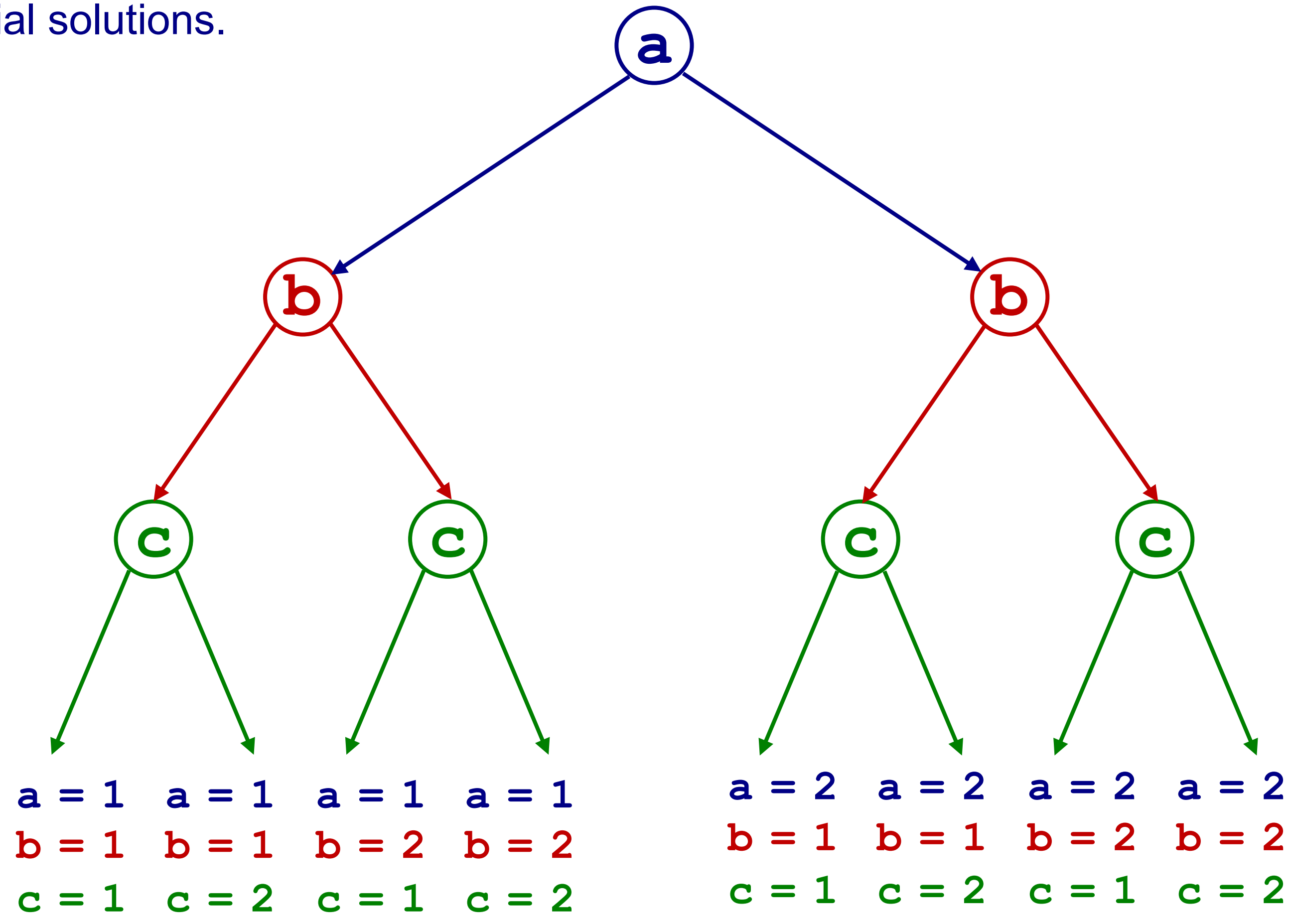
- These problems can be divided in, broadly, two main types. Informally,
- **Satisfaction Problems**
  - Finding a solution for a problem requires the exploitation of an exponential number of possibilities, but checking whether a proposed solution is correct can be done in polynomial time.
- **Optimization Problems**
  - Not only finding a solution for a problem requires the exploitation of an exponential number of possibilities, but checking whether a proposed solution is correct also requires an exponential time.
- More formally, the first type of problems are known to be NP-complete, whereas the second are known to be NP-Hard (there are many different classes for classifying these combinatorial problems – but we will not address them here*).

Note: See the classic book by Garey and Johnson, “Computers and Intractability: A Guide to the Theory of NP-Completeness”, or the link below for a quick introduction to the topic

https://en.wikipedia.org/wiki/Computers_and_Intractability

Combinatorial Problems

- The combinatorial nature of these problems, requiring non-deterministic search can be illustrated by the following example, where 3 binary choices are required, with **2^3** potential solutions.



Combinatorial Problems: Graph Algorithms

- Because of this combinatorial explosion, specialised techniques and languages have been proposed to deal with such complexity, although the basis problem remains.
- In general, the problems can only be solved for a relatively small size of n, although specialised algorithms, techniques and languages may push forward, significantly, the limit on the size of n that can be solved in a reasonable time.
- Here we will simply address a naïf approach to solve these problems and assess, experimentally, their efficiency in terms of **cpu time**.
- We illustrate the algorithms with the TSP problem, and leave the k-colouring problem as an exercise.
- **TSP (Travelling Salesperson Problem):**
  - **Satisfaction:** Find a tour that visits all nodes of a graph, not repeating any of the nodes that does not exceed a given length.
  - **Optimization:** Find the shortest tour that visits all nodes of a graph, not repeating any of the nodes (except the first and last node).

TSP Problem

- To solve this type of problems it is convenient to consider a recursion approach, where the problems to be solved have a decreasing complexity, until they become trivial.
- In this case, we consider two lists (arrays) of nodes:
  - Those that have already been visited in the tour (Visited)
  - Those that have not been visited yet. (ToVisit)
- On each step, a new node is moved from the ToVisit set, to the Visited step
  - Hence the problem becomes increasingly simpler, as there are less nodes to chose from, until there is no more choices to be made.
- As with any recursive function, the first thing to do is to check whether the recursion should be ended (trivial problem);
- Otherwise solve a simpler problem and use the solution of the simpler problem to obtain the solution of the larger problem.
- We exemplify this technique below with the optimisation version of the problem.

TSP Problem: Optimisation

- In addition to the **Visited** and **ToVisit** nodes and graph **G**, the function receives as arguments the cost of the path already done, **currCost,** and the cost of the best solution already found, **bestCost**(only better solutions are reported).
- If the problem is trivial (no more modes to visit, and there is an arc between the last and first modes) the function simply adds the cost of the arc between the last visited node and the first to close the tour, and returns the tour and its cost, if better.

```
def tsp_core(G, Visited, ToVisit, currCost, bestCost):
    """ Returns the Hamiltonian circuit (TSP) of minimum cost of a grap
    G, starting with a path through the Visited nodes, with a currCost,
    and finishing by the node ToVisit. Only solutions with a cost less
    than bestCost are returned. ""”
    if len(ToVisit) == 0:
        last_arc = G[Visited[-1]][Visited[0]]
            if last_arc < 0 or currCost + last_arc >= bestCost:
                return ([], math.inf)
            else:
                return (Visited, currCost + last_arc)
     else:
        ...
```

TSP Problem: Optimisation

- Otherwise,
  - each node in the **ToVisit** list, is selected for the **next_arc** node, if there is one;
  - a best tour is obtained (recursively), after updating the values of the parameters: newVisited (**T**) and newToVisit (**T**), the new current cost (**currCost + next_arc**), as well as the **bestCost**, updated in the loop.
  - among all the tours with each of the next arcs, the overall best tour is eventually returned as the optimal solution.

```
else:
   bestTour = []
   for j in ToVisit:
       next_arc = G[Visited[-1]][j]
       if next_arc > 0:
           V = Visited +[j]
           T = ToVisit.copy()
           T.remove(j)
           (cost,tour) = tsp_core(heur,G,V,T,currCost+next_arc,bestCost)
           if cost < bestCost:
                (bestCost, bestTour) = (cost, tour)
   return (bestCost, bestTour)
```

TSP Problem: Satisfaction

- We may now adapt this technique to obtain the satisfaction version of the problem. In fact all that is needed is to return a satisfactory solution once it has been found.
- We can do this by adding an extra parameter, satCost, and return immediately a solution with a cost no worse than this cost
- **Note**: Optimisation is obtained with satCost = 0. Why?

```
def tsp_core(G, Visited, ToVisit, currCost, bestCost, satCost):
    """ Returns the Hamiltonian circuit (TSP) of minimum cost of a graph
    G, starting with a path through the Visited nodes, with a currCost,
    and finishing by the node ToVisit. Only solutions with a cost less
    than bestCost are returned. Any solution with a cost less than
    satCost is immediately returned."""
   ...
   for j in ToVisit:
       ...
          (cost,tour) = tsp_core(heur,G,V,T,currCost+next_arc,bestCost)
          if cost < satCost:
              return (cost, tour)
          if cost < bestCost:
              (bestCost, bestTour) = (cost, tour)
   return (bestCost, bestTour)
```

TSP Problem: Heuristic Search

- In combinatorial problems it often pays off to follow some heuristic regarding the best decision to make.
- In this case, an intuitive heuristic to decide the next node to visit, is to select, among the nodes not visited yet, the node that is closer to the last node.
- The adaptation of the satisfaction version of the previous program is straightforward.
- Instead of visiting the nodes of the vector ToVisit in increasing order, one may previously sort the vector ToVisit, regarding the distances to the last node.
- Now, the selection of increasing order of the indices correspond to an increased order of the distances to the last node, as intended.

TSP Problem: Heuristic Search

- We may now adapt this technique to obtain the heuristic search version of the problem. As explained all that is needed is to sort the ToVisit nodes before selecting them for the next_arcs.
- We can do this by adding an extra parameter, heur. The nodes to visit are the node in ToVisit, either in their original order, or sorted by distance to the last node.

```
def tsp_core(heur, G, Visited, ToVisit, currCost, bestCost, satCost):
    """ Returns the Hamiltonian circuit (TSP) of minimum cost of graph G,
    starting with a path through the Visited nodes, with a currCost, and
    finishing with nodes ToVisit. Only circuits of cost less than bestCost
    are returned. Any solution with a cost less than satCost is immediately
    returned. Parameter heur specifies Heuristics search. """
    ...
    else:
        if heur:
            PrVisit = sortNext(Visited[-1], ToVisit, G)
        else:
            PrVisit = ToVisit
        bestTour = []
        for j in PrVisit:
            next_arc = G[Visited[-1]][j]
            ...
```

TSP Problem: Heuristic Search

- Sorting the nodes in ToVisit may be done by iteratively insertion the nodes into an empty list, in increasing order of their distances to the last Visited node.
- Note that nodes that are not connected are not inserted.
- As explained all that is needed is to sort the ToVisit nodes before selecting them

```
def sortNext(i, ToVisit, G):
   Next = [];
   for j in ToVisit:
      Next = insertSort(j, Next, i, G)
   return Next

def insertSort(j, Next, i, G):
    if G[i][j] <= 0:
        return Next
    else:
        n = len(Next)
        k = 0
        while k < n and G[i][Next[k]] < G[i][j]:
            k = k + 1
        return Next[0:k]+ [j] + Next[k:]
```

NP Problems: Assessing Performance

- Given the exponential nature of combinatorial NP problems, we may be interested in assessing the efficiency of our implementation.
- Top do so, we may “wraps” the previously defined function **tsp_core** in another function, **tsp**, that times the execution with calls to the time library, both before nd after the execution of **tsp_core**.

```
def tsp(G, heur, satCost):
    tStart = time.time()
    n = len(G)
    Visited = [0]
    ToVisit = [i for i in range(1,n)]
    cCost = 0                # current cost
    sCost = satCost          # sat cost
    bCost = math.inf         # best cost
    Tour = tsp_core(heur,G,Visited,ToVisit,cCost,bCost,sCost)
    tStop = time.time()
    elapsed = tStop - tStart
    return (round(elapsed, 2), Tour)
```

- Function tsp can be tested with any of the graphs in the graphs.zip files, and in different modes:
  - **heur**: True or False
  - **satCost**: 0 for optimisation