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 2017/2018

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

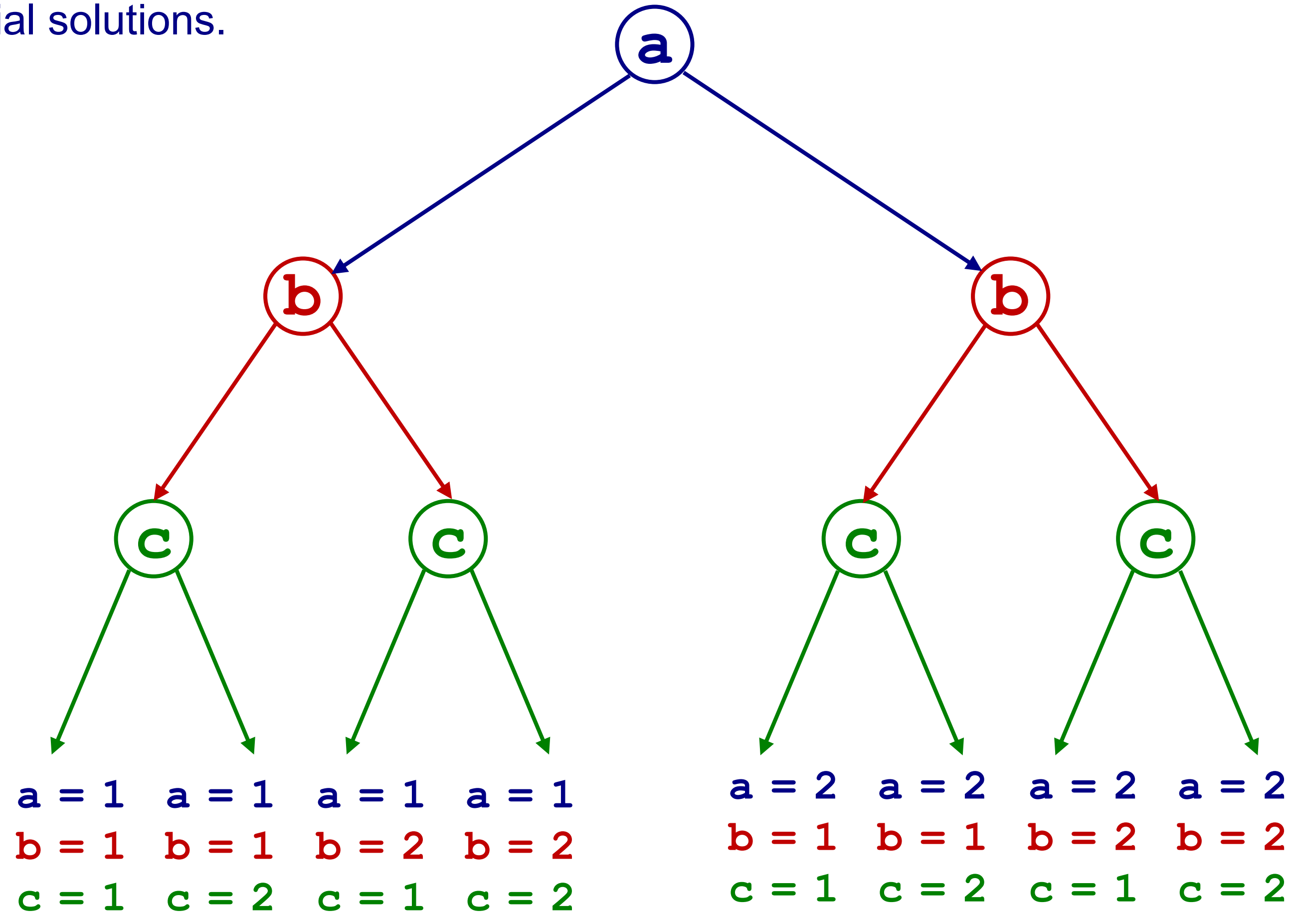
- These problems can be divided in, broadly, two main types. Informally,
- **Satisfaction Problems**
  - Finding a solution for a problem requires the exploitation of an exponential number of possibilities, but checking whether a proposed solution is correct can be done in polynomial time.
- **Optimization 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.

TSP Problem: Optimisation

- We exemplify this technique below with the optimisation version of the problem, that is easier to program.
- In addition to the **Visited** and **ToVisit** nodes, the graph, **G**, and the length of the path already done, **CostSoFar**, are parameters of the function.
- If the problem is trivial (no more modes to visit) the function simply adds the cost of the arc between the last visited node and the first to close the tour, and returns both the tour and its cost.

```
function [Tour,Cost] = hamilton_opt(G,Visited,CostSoFar,ToVisit)
   if length(ToVisit) == 0
      Tour = Visited;
      Cost = CostSoFar + G(Visited(end), Visited(1));
   else
      ... % solve the problem from a simpler one
   end
end
```

TSP Problem: Optimisation

- Otherwise,
  - each node in the **ToVisit** list, is selected in sequence as the **next** node,
  - a best tour is obtained (recursively), after updating the values of the parameters: newVisited (**NewVST**) and newToVisit (**NewTVS**) nodes, as well as the new CostSoFar (**NewCSF**)
  - among all the best tours with each of the next nodes, the overall best tour is selected and returned as the optimal solution.

```
Cost = Inf; Tour = [];
for i = 1:length(ToVisit)
   last = Visited(end);
   next = ToVisit(i);
   NewCSF = CostSoFar + G(last, next);
   NewVST = [Visited,next];
   NewTVS = [ToVisit(1:i-1),ToVisit(i+1:end)];
   [T,C] = hamilton_opt(G,NewVST,NewCSF,NewTVS);
   if C < Cost
      Tour = T;
      Cost = C;
   end
end
```

TSP Problem: Optimisation

- The complete algorithm is shown below:

```
function [Tour,Cost] = hamilton_opt(G,Visited,CostSoFar,ToVisit)
   if length(ToVisit) == 0
      Tour = Visited;
      Cost = CostSoFar + G(Visited(end), Visited(1));
   else
      Cost = Inf; Tour = [];
      for i = 1:length(ToVisit)
         last = Visited(end);
         next = ToVisit(i);
         NewCSF = CostSoFar + G(last, next);
         NewVST = [Visited,next];
         NewTVS = [ToVisit(1:i-1),ToVisit(i+1:end)];
         [T,C] = hamilton_opt(G,NewVST,NewCSF,NewTVS);
         if C < Cost
            Tour = T;
            Cost = C;
         end
      end
   end
end
```

TSP Problem: Satisfaction

- We may now adapt this technique to obtain the satisfaction version of the problem.
- In addition to the previous ones, the maximum Cost, **m**, of the accepted tours is the list of input parameters.
- As before, the trivial problem is checked first (no more nodes to visit).
- If the overall cost is longer than the maximum accepted cost, then the returned cost is **inf** (we consider a tour of infinite length as a no tour, i.e. an unacceptable solution).

```
function [Tour,Cost] = hamilton_sat(G,Visited,CostSoFar,ToVisit,m)
   if length(ToVisit) == 0
      Tour = Visited;
      Cost = CostSoFar + G(Visited(end), Visited(1));
      if Cost > m
         Cost = Inf;
      end
   else
      ... % solve the problem from a simpler one
   end
end
```

TSP Problem: Satisfaction

- Otherwise,
  - the recursive call is done as before, but only until an acceptable solution (i.e. one with a Cost less or equal to the maximum cost acceptable) is found.
  - Hence the loop is done with a WHILE instruction with an additional test on the cost (to check whether it is worth to continue).

```
Cost = Inf;
Tour = [];
i = 0;
while i < length(ToVisit) && Cost > m
    i = i + 1;
    last = Visited(end);
    next = ToVisit(i);
    NewCSF = CostSoFar + G(last, next);
    NewVST = [Visited,next];
    NewTVS = [ToVisit(1:i-1),ToVisit(i+1:end)];
    [T,C] = hamilton_sat(G,NewVST,NewCSF,NewTVS,m);
    if C < m
        Tour = T;
        Cost = C;
    end
end
```

TSP Problem: Satisfaction

- The complete algorithm is shown below:

```
function [Tour,Cost] = hamilton_sat(G,Visited,CostSoFar,ToVisit,m)
   if length(ToVisit) == 0
      Tour = Visited;
      Cost = CostSoFar + G(Visited(end), Visited(1));
      if Cost > m Cost = Inf; end
   else
      Cost = Inf; Tour = [];
      i = 0;
      while i < length(ToVisit) && Cost > m
         i = i + 1;
         last = Visited(end);
         next = ToVisit(i);
         NewCSF = CostSoFar + G(last, next);
         NewVST = [Visited,next];
         NewTVS = [ToVisit(1:i-1),ToVisit(i+1:end)];
         [T,C] = hamilton_sat(G,NewVST,NewCSF,NewTST, m);
         if C < Cost
            Tour = T; Cost = C;
         end
      end
   end
end
```

TSP Problem: Satisfaction with Heuristics

- 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 follows, with the same input parameters.

```
function [Tour,Cost] = hamilton_sat_h(G,Visited,CostSoFar,ToVisit,m)
   if length(ToVisit) == 0
      Tour = Visited;
      Cost = CostSoFar + G(Visited(end), Visited(1));
      if Cost > m
         Cost = Inf;
      end
   else
      ... % solve the problem from a simpler one
   end
end
```

TSP Problem: Satisfaction with Heuristics

- Otherwise,
  - As before, the recursive call is only done until an acceptable solution is found.
  - However, the the next node is not chosen is sequence (e.g. **i+1** as before) but selected by function **nextBest**/4.
  - Moreover, the NewTVS is updated, by **removing** the selected node.

```
Cost = Inf;
Tour = [];
i = 0;
while i < length(ToVisit) && Cost > m
    i = i + 1;
    last = Visited(end);
    next = nextBest(i,last,ToVisit,G);
    NewCSF = CostSoFar + G(last, next);
    NewVST = [Visited,next];
    NewTVS = remove(next, ToVisit);
    [T,C] = hamilton_sat_h(G,NewVST,NewCSF,NewTVS,m);
    if C < m
        Tour = T;
        Cost = C;
    end
end
```

TSP Problem: Satisfaction with Heuristics

- For completion we show below a possible implementation of functions **remove/2** and **nextBest/4**, used in the heuristic algorithm for the satisfaction problem.

- The **remove** function builds a new List from the last one, copying all the values except the node to be removed.

```
function NewList = remove(node, List)
   NewList = [];
   for v = List
      if v != node NewList = [NewList,v];  end
   end
end
```

- The function **nextBest**, sorts the nodes of the ToVisit list, according to their distance to the last visited node, and selects the **i**$^{\mathbf{th}}$ closest node.

```
function next = nextBest(i,last,ToVisit,G)
   Next = sortNext(G, last, ToVisit);
   next = Next(i);
end
```

TSP Problem: Satisfaction with Heuristics

- For sorting the nodes according to their distance to the last node, we may use the function below, that adapts the standard bubble sort function.

```
function Next = sortNext(G,last,List)
   n = length(List);
   Next = List;
   for k = n:-1:2
      for i = 1:k-1
         if G(last,Next(i)) > G(last, Next(i+1))
            x = Next(i);
            Next(i) = Next(i+1);
            Next(i+1) = x;
         end
      end
   end
end
```

TSP Problem: Satisfaction with Heuristics

- The algorithm (without the **remove** and **nextBest** functions) is shown below:

```
function [Tour,Cost] = hamilton_sat(G,Visited,CostSoFar,ToVisit,m)
   if length(ToVisit) == 0
      Tour = Visited;
      Cost = CostSoFar + G(Visited(end), Visited(1));
      if Cost > m Cost = Inf; end
   else
      Cost = Inf; Tour = [];
      i = 0;
      while i < length(ToVisit) && Cost > m
         i = i + 1;
         last = Visited(end);
         next = nextBest(i,last,ToVisit,G);
         NewCSF = CostSoFar + G(last, next);
         NewVST = [Visited,next];
         NewTVS = remove(next, ToVisit);
         [T,C] = hamilton_sat_h(G,NewVST,NewCSF,NewTVS,m);
         if C < m
            Tour = T; Cost = C;
         end
      end
   end
end
```

NP Problems: Assessing Performance

- Intuitively, we may expect a solution to the **optimisation** problem to be harder to find than that of a **satisfaction** problem (if the satisfaction goal is not too difficult), since we only need to find one solution, not comparing all the solutions
- Moreover, we may also expect that the **heuristic** version of the satisfaction algorithm is more efficient than the "sequential" one, if our heuristic is adequate.
- But how can we measure this efficiency?
- In practice, what can be done is to use a *system defined function* to measure the time that it took to execute a function. All programming languages have some such function. In Matlab function

```
[total, user, system] = cputime();
```

returns the total / user / system time elapsed since the start of the Matlab IDE (time is given in seconds).

- Hence to measure the time this function must be called **before** and **after** execution and the execution time is the **difference** between the returned results.

NP Problems: Assessing Performance

- We show below an example of assessing execution with the heuristic satisfaction version of the TSP (the other versions are similar).
- The function shown simply "wraps" the previously defined **hamiltonian** functions with calls to the cputime function, so as to return, not only the values of the tours and their costs, but also the elapsed user time, i.e. that used by the program itself to execute

```
function [Tour,Cost,time] = tsp_sat_h(G, maxCost);
   [t0, u0, s0] = cputime();
   n = size(G,1);
   [Tour,Cost] = hamilton_sat_h(G,[1],0,2:n,maxCost);
   [t1, u1, s1] = cputime();
   time = u1-u0;
end
```

NP Problems: Assessing Performance

- Below we show results obtained with the different versions of the TSP applied to graph in file “graph_10_90.txt”, previously read into the adjacency matrix **M10**

```
>> [Tour,Cost,time] = tsp_opt(M10)
   Tour = 1 3 6 5 9 10 2 8 7 4
   Cost =  201
   time =  115.17
>> [Tour,Cost,time] = tsp_sat(M10, 210)
   Tour = 1 3 6 5 9 10 2 8 7 4
   Cost =  201
   time =  19.027
>> [Tour,Cost,time] = tsp_sat_h(M10, 210)
   Tour = 1 3 6 5 9 10 2 8 7 4
   Cost =  201
   time =  3.3025
>> [Tour,Cost,time] = tsp_sat(M10, 220)
   Tour = 1 3 2 5 9 10 8 7 6 4
   Cost =  219
   time =  12.917
>> [Tour,Cost,time] = tsp_sat_h(M10, 220)
   Tour = 1 3 2 5 9 10 8 7 6 4
   Cost =  219
   time =  0.49499
```