EDAF05

Algorithms

Stable Matching problem

Is the problem of finding a stable matching between two equally sized sets of elements given an ordering of preferences for each element. A matching is a mapping from the elements of one set to the elements of the other set. A matching is not stable if: * There is an element A of the first matched set which prefers some given element B of the second matched set over the element to which A is already matched, and * B also prefers A over the element to which B is already matched. In other words, a matching is stable when there does not exist any match (A, B) by which both A and B would be individually better off than they are with the element to which they are currently matched.

Greedy Algorithms

Greedy problems can be solved by taking the best option first and do so until the problem is solved, This solves some problems optimal but far from all. An example of this is the spanning USA lab in the course. During this lab you want to create a minimum spanning tree. This is done by sorting the weights in the graph and choosing the least costly edge while avoiding cycles. This will result in a optimal solution.

Dijkstra

Given a graph G = (V,E) we want to find the shortest path (or cheapest) from a given start node to an end node. The Dijkstra algorithm is designed to find the optimal solution but with the constant that all costs or distances has to be a positive integer.

The Algorithm:

```
node n = (value,
def Dijkstra(G,n,target):
    n.value = 0; #sets the start nodes value to 0, All other nodes should have inf as value.
    Visited = {}
    while n != target:
        for Eedges connected to n not in Visited:
            if n.value + edge.value < edge.end.value:
                edge.end.value= n.value + edge.value
        Visited.append(n)
        n = nextNode #pickout a node not in visited connected to n
    return n.value
```

Prim's

O(e log v) where e is the edges and v is the vertices.

```
def prim(G,start): # start can be randomly selected
    pq = PriorityQueue()
        for v in G:
            v.setDistance(sys.maxsize)
            v.setPred(None)
    start.setDistance(0)
        pq.buildHeap([(v.getDistance(),v) for v in G])
        while not pq.isEmpty():
            currentVert = pq.delMin()
            for nextVert in currentVert.getConnections():
                newCost = currentVert.getWeight(nextVert)
                if nextVert in pq and newCost<nextVert.getDistance():
                        nextVert.setPred(currentVert)
                        nextVert.setDistance(newCost)
                        pq.decreaseKey(nextVert,newCost)
```

Kruskal's

O(e log v) where e is the edges and v is the vertices. Given a Graph G = (V,E). We want to find the minimum spannig tree.

```
def kruskal(V,E):
sort(E) //Sorts E, low to high
forest = {}
while E.length != 0:
```

```
    edge = E.pop()
    if edge has no endpoint in the forest:
        forest.append(edge)
return forest
```

Simple and brilliant!

Graph connectivity

A graph is connected when there is a path between every pair of vertices. In a connected graph, there are no unreachable vertices. A graph that is not connected is disconnected. A graph G is said to be disconnected if there exist two nodes in G such that no path in G has those nodes as endpoints. A graph with just one vertex is connected. An edgeless graph with two or more vertices are disconnected. It is closely related to the theory of network flow problems.

Bipartite graphs - A bipartite graph is a graph whose vertices can be divided in two disjoined sets such that every edge connects two vertices, one in each of the disjoint sets.

Depths-first search (DFS)

DFS is a search algorithm designed to find a specified value or node in a Graph. One starts at the root and explores as far as possible along each branch before backtracking. This algorithm is suitable for finding items belived to be in the lower leaves. Complexity O(V + E).

A Recursive implementation:

```
def DFS(G,v):
    label v as discovered
    for all edges from v to w in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G,w)
```

A non-recursive implementation:

```
def DFS(G,v):
    let S be a stack
        S.push(v)
        while S is not empty
            v = S.pop()
        if v == target :
            return target

            if v is not labeled as discovered:
                    label v as discovered
                    for all edges from v to w in G.adjacentEdges(v) do
                        S.push(w)
    return -1
```

Breadth-first search (BFS)

BFS is an alternative to the DFS algorithm. It differs by exploring the neighbor nodes first, before moving to the next level neighbors instead of searching from the bottom up as the DFS does.

Complexity O(V+E)

```
def BFS(G,v,target):
    create empty set Visited
    Q = new Stack()
        Q.append(v)
        while Q is not empty:
            current = Q.pop()
            if current.value == target:
                    return current
            for each node n that is adjacent to current:
                    if n is not in Visited:
                    Q.enqueue(n)
```

Divide and Conquer

Is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions

to the sub-problems are then combined to give a solution to the original problem. An example of this is the Fibonacci series, to find the n:th number in the series you have to know all the previous numbers in the series.

```
#find the n:th number in the Fibonacci sequence.
def fibbonacis(number n):
    if n =< 1:
        return fibbonacis(n-1) + fibbonacis(n-2)
    else:
        return 1
```

Or a search algorithm to find the largest perfect subtree in a binary search tree.

```
#finding the largest subtree in a binary searchtree
def func(node n):
    if(n!=None):
        tuple = (func(n.left),func(n.right))
        MyMax = 1+ min(tuple[0],tuple[1])
        TotalMax = max(MyMax,tulp[0],tuple[1])
        return (myMax,TotalMax);
    else:
        return (0,0)
```

Dynamic programming

Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution. The technique of storing solutions to subproblems instead of recomputing them is called "memoization".

Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. In comparison, a greedy algorithm treats the solution as some sequence of steps and picks the locally optimal choice at each step. Using a greedy algorithm does not guarantee an optimal solution, because picking locally optimal choices may result in a bad global solution, but it is often faster to calculate. Some greedy algorithms (such as Kruskal's or Prim's for minimum spanning trees) are however proven to lead to the optimal solution.

Let's revisit the Fibonacci sequence. The previous code example is enormously memory costly due to the recursion and in python we would not be able to find large sequence numbers due to the compiler, but with dynamic programming we will have a O(n) time and space complexity.

I mean it isn't the most brilliant thing in the world but it is an easy example to understand. The difference to the previous example is that now we memorize the previous results to the array and use the array to calculate the next number, the same thinking applies to all dynamic programming problems.

```
def fibbonaci(n):
    array ={}
    array[0] = 0
    array[1] = 1
    for i in range(2,n):
        array[i] = array[i-1] + array[i-2]

    return array
```

Network flow

Network flow problems are very similar to graph problems but has one main difference, all edges have weights. Network flow is a problem that consists of a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge. A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, unless it is a source, which has only outgoing flow, or sink, which has only incoming flow. This algorithm can be used to model traffic flow in cities, sewer systems or network capacity.

The max-flow min-cut theorem states that in a flow network, the maximum amount of flow passing from the source to the sink is equal to the total weight of the edges in the minimum cut, i.e. the smallest total weight of the edges which if removed would disconnect the source from the sink.

Examples:

suppose the edges are roads with cars, pipes with water or wires with electric current. flow represents the volume of water allowed to flow though the pipes, the number of cars the roads can sustain in traffic and net electric current. Effectiively, it's the "bottleneck" value for the amount of flow that can pass though the network from source to sink under all constraints.

1. Number Of maximum students to transfer between cities.
2. matching between job applicants and companies.
3. How many Domino rocks can fit on a chess board filled with obstacles.

Fulkerson-Ford

Fulkerson-Ford is a greedy-algorithm to the maximum-flow minimum-cut problem. It works by pushing flow up all edges until it can't. When the algorithm can't push any more flow in the graph we have our maximum flow in the network.

Complexity O(E max|f|) or O(E+V). ((Can be done in O(VE) by Orlins algorithm, not covered in the course.))

```
def Ford-Fulkerson(G,source,sink):
    flow = 0
    for each edge(u, v) in G:
        flow(u, v) = 0
    while there is a path, p, from s -> t in residual network G_f:
        residual_capacity(p) = min(residual_capacity(u, v) : for (u, v) in p)
        flow = flow + residual_capacity(p)
        for each edge (u, v) in p:
            if (u, v) is a forward edge:
                flow(u, v) = flow(u, v) + residual_capacity(p)
            else:
                flow(u, v) = flow(u, v) - residual_capacity(p)
    return flow
```

Special cases

Maximum Edge-disjoint path

Given a directed graph G = (V, E) and two vertices s and t, we are to find the maximum number of edge-disjoint paths from s to t. We do this by giving all the edges in the graph (expect the one(s) leading from the source or to the sink) the maximum capacity of one unit.

Node independent path

Given a directed graph G = (V, E) and two vertices s and t, we are to find the maximum number of independent paths from s to t. Two paths are said to be independent if they do not have a vertex in common apart from s and t.

Complexity

Big O notation

Big O notation is used to classify algorithms according to how their running time or space requirements grow as the input size grows. For example, the time (or the number of steps) it takes to complete a problem of size n might be found to be O(n) = 4n^2 — 2n + 2. As n grows large, the n^2 term will come to dominate, so that all other terms can be neglected—for instance when n = 500, the term 4n^2 is 1000 times as large as the 2n term. Ignoring the latter would have negligible effect on the expression's value for most purposes. Further, the coefficients become irrelevant if we compare to any other order of expression, such as an expression containing a term n^3 or n^4. Even if T(n) = 1,000,000n^2, if U(n) = n^3, the latter will always exceed the former once n grows larger than 1,000,000 (T(1,000,000) = 1,000,0003= U(1,000,000)). Notice that we use T(n) and U(n), these can be reduced to a big O notation, which would be (T(n)) O(n)= n^2 and (U(n)) O(n) = n^3. T(n), denoting the exact time needed to calculate the data of size n. It's very useful when calculate the time needed of a recursive function.

Memory complexity

Big-O order of memory means how does the number of bytes needed to execute the algorithm vary as the number of elements processed increases. In your example, I think the Big-O order is n squared, because the data is stored in a square array of size nxn. Usually mentioned in dynamic programming.

P, NP, NP-Complete & NP-hard

P != NP

P != NP is one of the millennium problems. The problem is essentially to prove that some problems can't be computed in polynomial time. It is not proven that p != NP but it is a guess made by most computer scientists. To prove that some problems can be solved in polynomial time would essentially create mass panic because all computer security is built on NP problems.

Decision problem

A decision problem is a problem that can be posed as a yes-no question of the input values.

P

P is a complexity class that represents the set of all decision problems that can be solved in polynomial time. That is, given an instance of the problem, the answer yes or no can be decided in polynomial time. All of the algorithms in this class is p problems.

NP

NP - nondeterministic polynomial time. NP is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time. This is functions which gets harder with size, like solving sudoku. sudoku with a 9x9 grid may relatively easy be solved by brute force but a 18 by 18 grid will take much more time and wont follow a polynomial time increase. An example of NP-problems being used in computer security is RSA-encryption. Where finding the keys by knowing the resulting modulo operation.

NP-Complete

NP-complete problems are in NP, the set of all decision problems whose solutions can be verified in polynomial time.

3-sat

Is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE.

Set Cover

Given a set of elements { 1 , 2 , . . . , n }(called the universe) and a collection S of m sets whose union equals the universe, the set cover problem is to identify the smallest sub-collection of S whose union equals the universe.

Analogy:

"Imagine being in a lego store and you want to build the Death Star. And me being an truly evil person has, besides leaving lego on the floor, displaced all pieces of the Death Star in all of the other boxes containing Death Star pieces, and vice versa. Being a truly lazy person you want to figure out the minimum amount of boxes to carry home that has the necessary pieces for building the Death Star. Which boxes do you pick?"

In the analogy the Death Star is the universe and the known boxes is the sets.

3-dimensional matching

Let X, Y, and Z be finite, disjoint sets, and let T be a subset of $X \times Y \times Z$. That is, T consists of triples (x, y, z) such that $x \in X$, $y \in Y$, and $z \in Z$. Now $M \subseteq T$ is a 3-dimensional matching if the following holds: for any two distinct triples $(x1, y1, z1) \in M$ and $(x2, y2, z2) \in M$, we have $x1 \neq x2$, $y1 \neq y2$, and $z1 \neq z2$. (Note that all three must be true at the same time, so (1,2,3) and (1,4,5) would not hold)

3-satisfiability

Given a set of boolean variables: x1, x2... We'll define a literal to be either a variable xi or NOT xi. A clause is defined as to be 3 literals OR'd together, e.g. (x1 OR (NOT x4) OR x5)

A 3-SAT problem is a "conjunction of clauses" of the form:

(x1 OR x2 OR x4) AND (x3 OR (NOT x4) OR x7) AND ...

Solving a 3-SAT problem is the act of finding a set of variable assignments to True or False that make that statement true or alternately providing a proof that no such set of variable assignments can exist.

Vertex cover

Given a Graph G = (V,E), all edges e in E has at least one end in the resulting set.

The vertex cover has a NP-hard problem as well, The Minimum vertex cover, This is the optimization problem of the vertex cover and it aims to find the smallest subset of vertices which can reach all vertices in the graph.

Graph colouring

Is a problem which consists of an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two connected vertices share the same color. It is important to note that the minimum amount of colors to make this a np-complete problem is 3.

On the other hand the Graph Coloring Optimization problem, which aims to find the coloring with minimum colors is np-hard, because even if you are given a coloring, you will not be able to say that it's minimum or not.

Hamiltonian path & Hamiltonian cycle

Given a graph G you have to check if there is a way of traversing the graph and visiting each node once. If such path is possible the graph contains a Hamiltonian path. A Hamiltonian cycle is a Hamiltonian path which starts and ends in the same node.

In the same way as the graph coloring problem the Hamiltonian cycle has a optimization problem: The Travelling salesman problem described below.

(Maximum) independent set

Given a graph G = (v,E), we say a set of nodes belonging to the a set S. We want to find the maximum set of nodes which are independent to one and another. Two nodes are independent if the nodes don't have an edge directly connecting them.

Travelling salesman

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

The travelling salesman problem is a generalization of the hamiltonian cycle problem.

knapsack problem

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The Knapsack problem as well as previous problems has a optimization problem which is NP-hard, if we want to find the optimal solution to the knapsack problem, i.e the perfect ratio between weight and value.

NP-hard

NP-hard is the defining property of a class of problems that are, informally, "at least as hard as the hardest problems in NP". Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems. A memory rule I have: "Is this problem similar to a NP-Complete problem but adds a constraint?", if the answer is yes we have a NP-Hard problem.

I want to be very clear: NP-hard and NP-complete are very much alike and the ONLY thing that differences is that NP-complete problems are decision problems. An example:

The knapsack problem is NP-complete when we ask the question: "given the maximum weight W and the items I={.....}, can we achieve a total value of n?", but when we ask the question "what is the optimal solution to the knapsack problem given the maximum weight and the set of items I={....}?".

Halting problem (not include in the course, just a fun fact)

The halting problem is a decision problem about properties of computer programs on a fixed Turing-complete model of computation, i.e., all programs that can be written in some given programming language that is general enough to be equivalent to a Turing machine. The problem is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations on the amount of memory or time required for the program's execution; it can take arbitrarily long, and use arbitrarily as much storage space, before halting. The question is simply whether the given program will ever halt on a particular input.

For example, in pseudocode, the program `while (true) continue` does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program

`print "Hello, world!"` does halt.

While deciding whether these programs halt is simple, more complex programs prove problematic. One approach to the problem might be to run the program for some number of steps and check if it halts. But if the program does not halt, it is unknown whether the program will eventually halt or run forever.

Certificate

A certificate is a string that certifies the answer to a computation, or certifies the membership of some string in a language. A certificate is often thought of as a solution path within a verification process, which is used to check whether a problem gives the answer "Yes" or "No".

If that doesn't make sense to you don't fret, a certificate is the solution your algorithm spits out. It must include all information necessary so that it can be verified that, in fact, it is a solution.

How to know what degree a problem is

The strategy is this: 1. Try to reduce your problem to a known P problem. if this can't be done move on to step 2. 2. Try to reduce a known NP-Complete problem, if this can't be done it is probably a NP-Hard problem.

By reducing I mean if you set a P,NP,NP-Complete problem as a lower bound and you can use your problem to solve the lower bound problem then your problem is at least as hard as the problem you use as a measuring stick. so if you can reduce your problem down to NP-Complete problem you either have a NP-Complete problem or a NP-Hard problem.

Author: Hampus Weslien D16, This is a fork from Måns Ansgariusson, C14 summary where if fixed some of spelling and grammar mistakes, as well as added a few more NP-hard problems.