

GREEDY ALGORITHMS



Fatima Jinnah
Women University

5/2/2021

Table of Contents

Definition of Algorithm:	2
Definition of Greedy Algorithm:	2
Characteristics of greedy algorithm	2
Feasible and optimal solution:	2
Greedy choice property	3
Structure of a Greedy Algorithm	5
Advantages/Disadvantages:	12
Applications:	12
Conclusion	12
References:.....	13

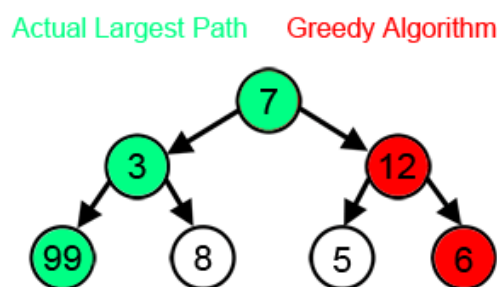
Definition of Algorithm:

“An **algorithm** is a step-by-step problem-solving method.”

Definition of Greedy Algorithm:

“A **greedy algorithm** is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.”

However, in many problems, a greedy strategy does not produce an optimal solution. For example, in the animation below, the greedy algorithm seeks to find the path with the largest sum. It does this by selecting the largest available number at each step. The greedy algorithm fails to find the largest sum, however, because it makes decisions based only on the information it has at any one step, and without regard to the overall problem.



Characteristics of greedy algorithm

- There is an ordered list of resources which contain profits, costs or value etc. These measure the constraints on a system.
- Maximum quantity of resources such as max profit, max values etc. are taken when a constraint applies.
- In fractional knapsack problem the maximum value is taken first according to availability of capacity.

Feasible and optimal solution:

Suppose the problem is to travel from A to B.

P: A-> B

Now there are many possibilities to travel from A to B. This distance can be covered by walk, by bike, by car, by train, by flight or any other mean. But there is a **constraint** or condition in the problem that you have to cover this in twelve hours only. Let that you cannot cover this by walk, by bike or by car in twelve hours so there are then only 2 possibilities left either by train or by flight.

There are many solutions to the problem but the solutions that satisfy the condition are called **Feasible Solutions**.

Next you have to cover this distance in minimum cost. Now this problem becomes a minimization problem. As out of two feasible solutions, I can travel the distance in minimum cost by train so this is the optimal solution.

So, the solution which is feasible and is also giving best results then it is **Optimal Solution**.

For any problem, there can be more than one feasible solution but there is only one optimal solution.

If a problem requires either minimum or maximum results then this type of problem is **Optimization Problem**.

Greedy choice property

A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

How do you decide which choice is optimal?

Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

For Example:

Being a very busy person, you have exactly T time to do some interesting things and you want to do maximum such things in the limited time that you have.

You are given an array A of integers, where each element a is a time that a thing takes to complete.

You want to calculate maximum such things that you can do in the limited time you have. This is a simple Greedy-algorithm problem.

In each iteration, you have to greedily select the things which will take the minimum amount of time to complete while maintaining two variables `currentTime` and `numberOfThings`. To complete the calculation you must:

- Sort the array A in increasing order.
- Select each to-do item one-by-one.
- Add the time that it will take to complete that to-do item into `currentTime`.

- Add one to numberOfThings

Repeat this as long as the currentTime is less than or equal to T.

Let **A** = {5, 3, 4, 2, 1} and **T** = 6

After sorting, **A** = {1, 2, 3, 4, 5}

After the 1st iteration:

- **currentTime** = 1
- **numberOfThings** = 1

After the 2nd iteration:

- **currentTime** is $1 + 2 = 3$
- **numberOfThings** = 2

After the 3rd iteration:

- **currentTime** is $3 + 3 = 6$
- **numberOfThings** = 3

After the 4th iteration, **currentTime** is $6 + 4 = 10$, which is greater than T. Therefore, the answer is 3.

Implementation:

```
#include "stdafx.h"

#include <iostream>

#include <algorithm>

using namespace std;

const int MAX = 105;

int A[MAX];

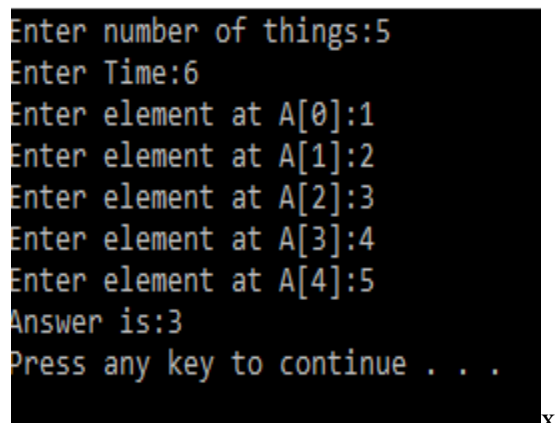
int main()

{
    int T, N, numberOfThings = 0, currentTime = 0;
    cout<<"Enter number of things:";
    cin >> N;
```

```

cout<<"Enter Time:";
cin>> T;
for(int i = 0;i < N;++i)
{
    cout<<"Enter element at A["<<i<<"]:" ;
    cin >> A[i];
}
sort(A, A + N);
for(int i = 0;i < N;++i)
{
    currentTime += A[i];
    if(currentTime > T)
        break;
    numberOfThings++;
}
cout << "Answer is:" <<numberOfThings << endl;
    system ("pause");
return 0;
}

```



The screenshot shows the execution of the provided C++ code. The user enters '5' for the number of things and '6' for the time. Then, they enter five elements for the array A: 1, 2, 3, 4, and 5. The program outputs 'Answer is:3' and prompts the user to 'Press any key to continue . . .'. A small 'x' icon is visible in the bottom right corner of the terminal window.

Structure of a Greedy Algorithm

Greedy algorithms take all of the data in a particular problem, and then set a rule for which elements to add to the solution at each step of the algorithm. In other words, greedy algorithms work on problems for which it is true that at every step, there is a choice that is optimal for the problem up to that step, and after the last step, the algorithm produces the optimal solution of the complete problem.

Greedy algorithms can be characterized as being 'short sighted', and also as 'non-recoverable'. They are ideal only for problems which have 'optimal substructure'. Despite this, for many simple problems (e.g. giving change), the best suited algorithms are greedy algorithms.

Examples:

Following are some standard algorithms that are Greedy algorithms.

1) Kruskal's Minimum Spanning Tree (MST):

Spanning Tree: A connected and undirected graph, a spanning tree of that graph is sub graph that is tree and connected to all vertices together.

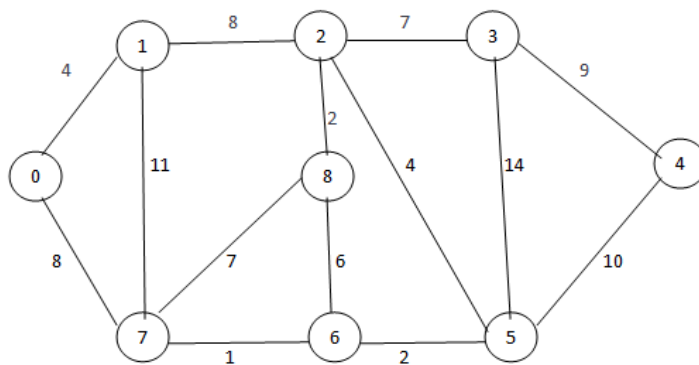
Minimum Spanning Tree: The spanning tree of a graph whose sum of weights of edges is minimum.

A graph may have more than one minimum spanning tree.

In Kruskal's algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.

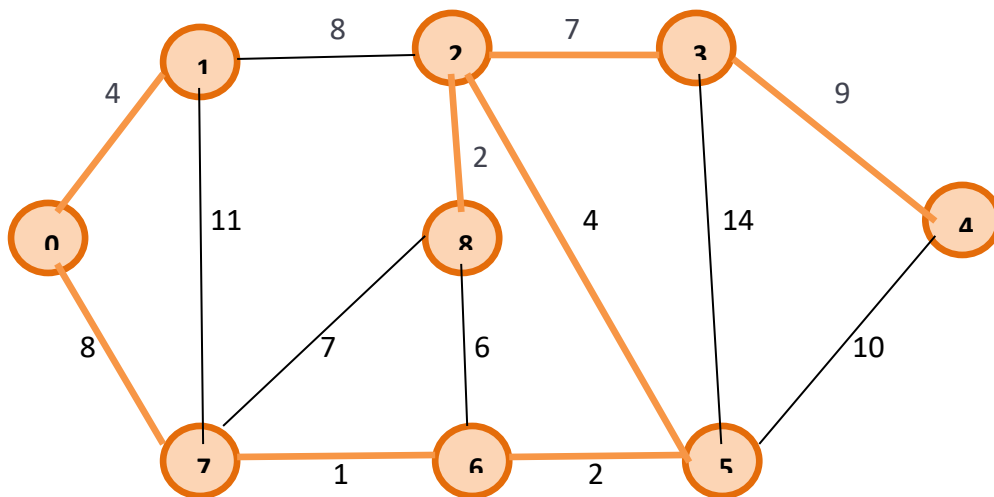
For example:

Consider the following graph



Edges	7,6	8,2	6,5	0,1	2,5	8,6	2,3	7,8	0,7	1,2	3,4	5,4	1,7	3,5
Weight	1	2	2	4	4	6	7	7	8	8	9	10	11	14

The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having $(9 - 1) = 8$ edges.

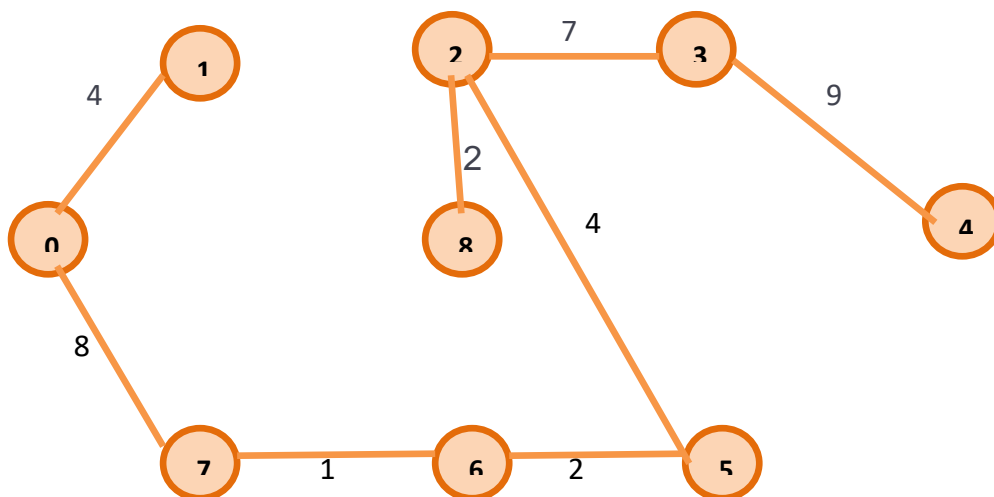


1. Pick edge 7-6, no cycle is formed so include it.
2. Pick edge 8-2, no cycle is formed so include it.
3. Pick edge 6-5, no cycle is formed so include it.
4. Pick edge 0-1, no cycle is formed so include it.
5. Pick edge 2-5, no cycle is formed so include it.
6. Pick edge 8-6, it will form a cycle so discard it.
7. Pick edge 2-3, no cycle is formed so include it.
8. Pick edge 7-8, it will form a cycle so discard it.
9. Pick edge 0-7, no cycle is formed so include it.
10. Pick edge 1-2, it will form a cycle so discard it.
11. Pick edge 3-4, no cycle is formed so include it.

Edges	7,6	8,2	6,5	0,1	2,5	2,3	0,7	3,4
Weight	1	2	2	4	4	7	8	9

Now we have 8 edges.

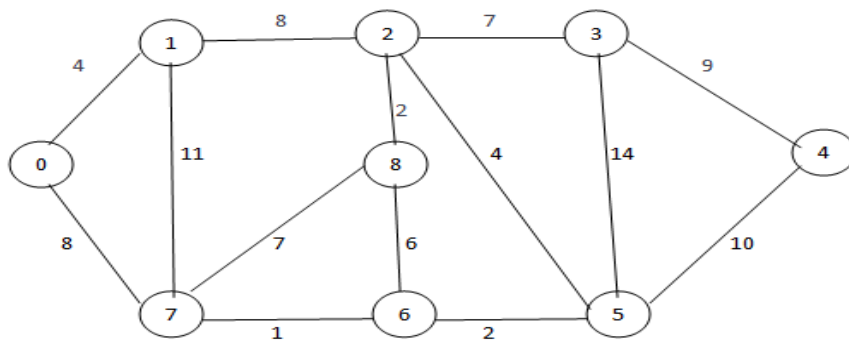
Now we have are Minimum Spanning Tree (MST)



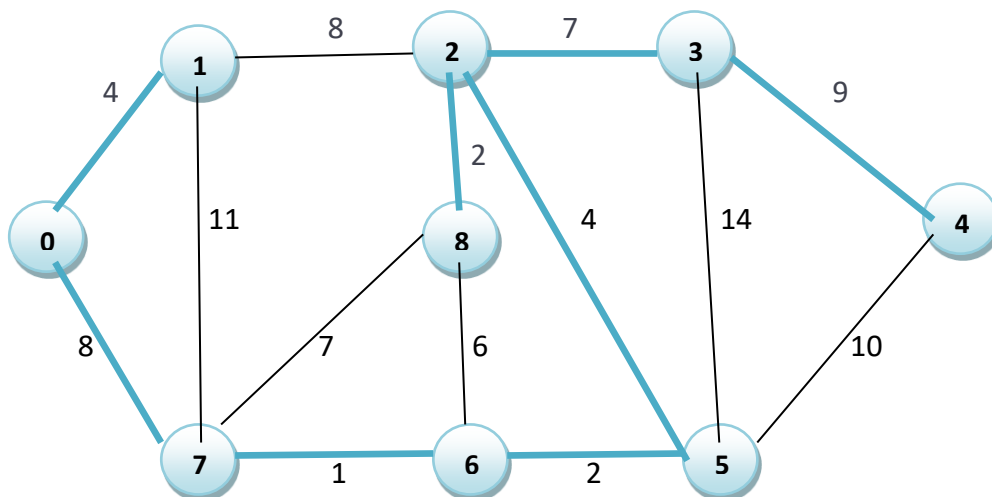
2) Prim's Minimum Spanning Tree:

In Prim's algorithm also, we create a MST by picking edges one by one. We maintain two sets: set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.

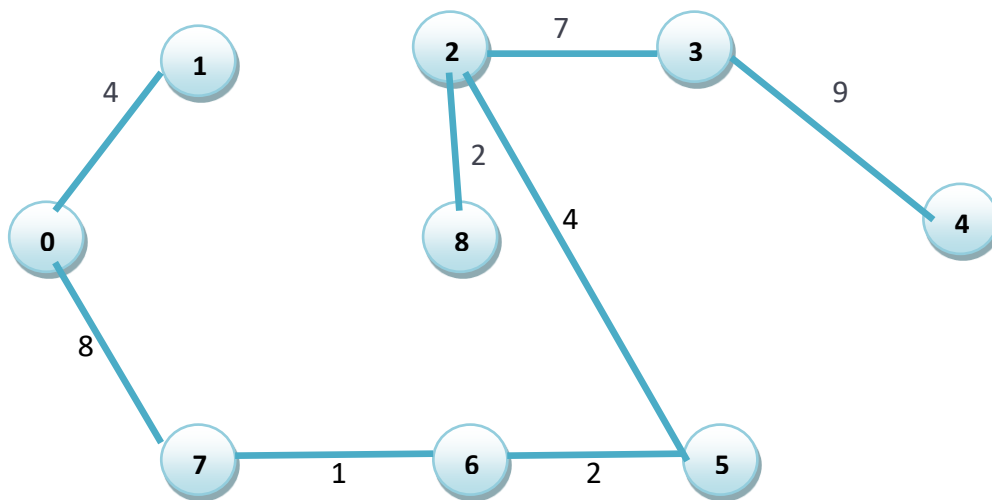
For example:



We will keep on picking the minimum key value and not already included in MST. If one vertex is picked we will update it in MST and the other vertex will be considered for next time if it is not already visited.



This is the MST

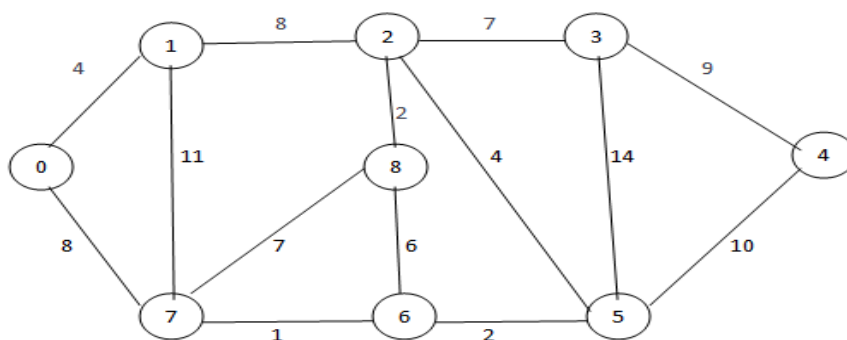


3) Dijkstra's Shortest Path:

The Dijkstra's algorithm is very similar to Prim's algorithm. The shortest path tree is built up, edge by edge. We maintain two sets: set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contain not yet vertices.

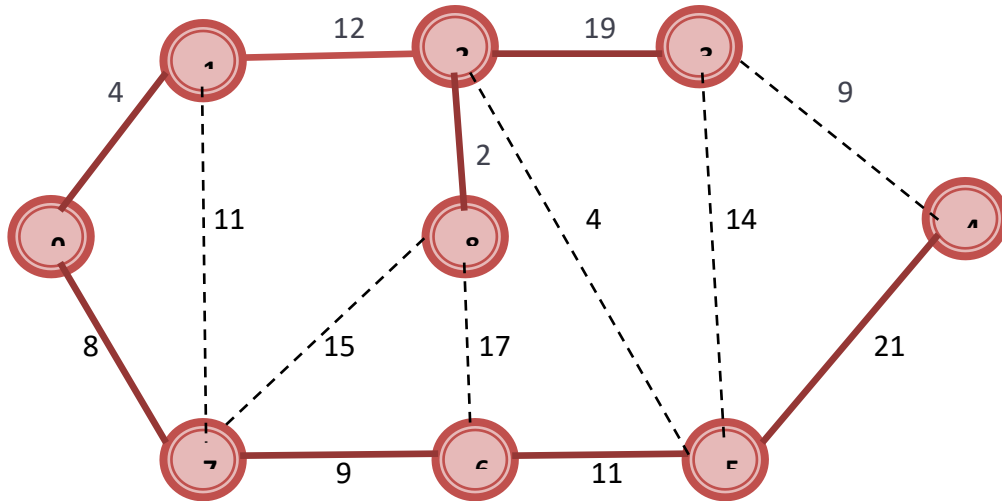
For example:

Consider the following graph

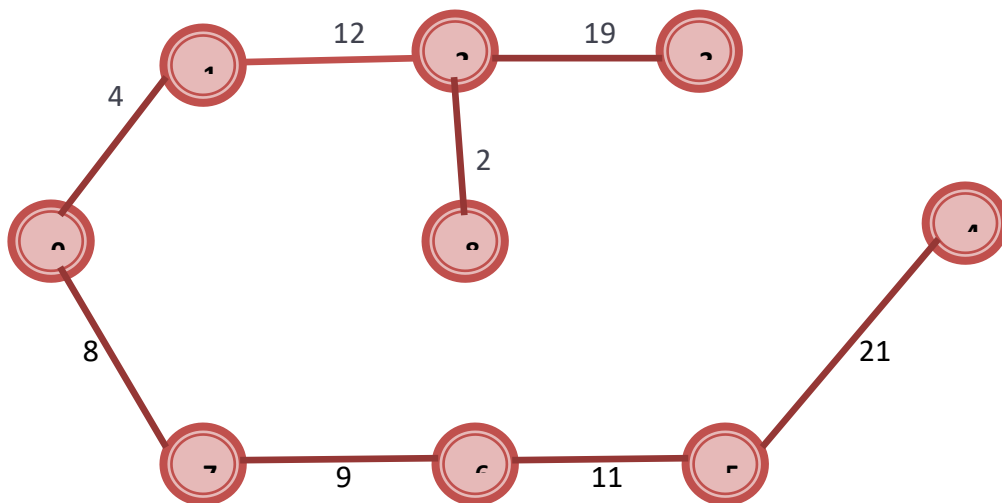


Now we will pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. The vertex 1 is picked and added to *sptSet* Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes

12. We repeat the above steps until *sptSet* does include all vertices of given graph.



This is the MST:



4) Huffman Coding:

Huffman Coding is a loss-less compression technique. It assigns variable length bit codes to different characters. The Greedy Choice is to assign least bit length code to the most frequent character. There are two parts of Huffman Coding

1. First one to create Huffman tree
2. Second one to traverse the tree to find codes

5) Algorithm for Insertion sort a_1, a_2, \dots, a_n : real numbers $n \geq 2$:

For $j := 2$ **to** n $i := 1$

While $a_j > a_i$

$i := i + 1$

$m := a_j$

for $k := 0$ **to** $j - i - 1$

$a_{j-k} := a_{j-k-1}$

$a_i := m$

{ a_1, \dots, a_n is in increasing order}

6) Procedure change (c_1, c_2, \dots, c_r : values of denominations of coins, where $c_1 > c_2 > \dots > c_r$; n : a positive integer)

For $i := 1$ **to** r

$d_i := 0$ { d_i counts the coins of denomination c_i used}

while $n \geq c_i$

$d_i := d_i + 1$ {add a coin of denomination c_i }

$n := n - c_i$

{ d_i is the number of coins of denomination c_i in the change for $i = 1, 2, \dots, r$ }

7) Greedy Algorithm for Scheduling Talks:

- **lets first arrange the lectures in the order of their finishing time**

1. **9:00-10:00**
2. **9:30-10:30**
3. **9:00-11:00**
4. **11:00-11:30**
5. **10:00-12:00**
6. **11:30-12:30**

We chosen lecture 1 first

We cannot add lecture 2 and 3 since lecture 1 overlap with them

We then chosen lecture 4. we cannot add lecture 5 since lecture 4 overlap with them

Finally we choose Lecture 6

So Chosen Lectures are: 9:00 - 10:00, 11:00 - 11:30 and 11:30 - 12:30

Advantages/Disadvantages:

Greedy algorithms have some advantages and disadvantages:

1. It is quite easy to come up with a greedy algorithm (or even multiple greedy algorithms) for a problem.
2. Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.
3. The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.
4. The execution rate of greedy algorithm is very low.

Applications:

Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum. If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods like dynamic programming. Using greedy routing, a message is forwarded to the neighboring node which is "closest" to the destination.

Conclusion

There is no hard and fast rule explaining when this proof strategy will succeed, just as there is no hard and fast rule explaining when “greedy stays ahead.” Exchange arguments function especially well because all optimal solutions are the same size and only vary in

cost, so you can explain how you're going to delete any aspect of the optimal solution and snap another one in its place. Determining when to use exchange arguments is a skill that you will develop as you work through the problem set.

References:

- Introduction to Algorithms (Cormen, Leiserson, and Rivest) 1990, Chapter 17 "Greedy Algorithms" p. 329
- Introduction to Algorithms 6.046J/18.401J LECTURE_16
- www.hackerearth.com/basics-of-greedy-algorithms
- www.wisdom.weizmann.ac.il/~feige/mypapers/ciac_greedy.pdf
- www.bowdoin.edu/~ltoma/teaching/cs231/spring16/Lectures/09
- <https://books.google.com.pk/books?isbn=3540424954> by Rizo Sakellariou, John Keane, John Gurd - 2001

THE END
