DS.GR.25

Shortest Path Algorithms

Input can be:

- a graph or a digraph
- weighted or unweighted

Let c(i,j) be the cost of traversing edge (vi,vj).

Then the path length of the path
P = {v1, v2, . . . , vn}

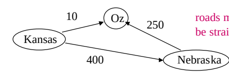
is $\sum_{i=1}^{n-1} c(i,i+1)$

where c(i,j) is the weight on edge (vi,vj) for a weighted graph and is just 1 for an unweighted graph.

DS.GR.26

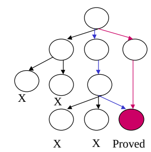
Examples:

- shortest route from one city to another



roads may not be straight!

- shortest number of steps to prove a theorem using a graph search technique.



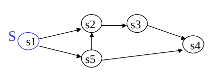
Shorter Proof

DS.GR.27

1. The Unweighted Shortest Path Algorithm

Let G be an unweighted digraph,
S be a start node in G,
Q be a queue of nodes to process,
T be a table with the following structure

Node	Known	Dist	Path
identifier	unused in this algorithm	distance from S	prior node on shortest path from S

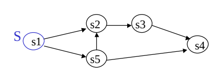


T

Node	Known	Dist	Path
s1	0	0	0
s2	0	∞	0
s3	0	∞	0
s4	0	∞	0
s5	0	∞	0

initial configuration

DS.GR.28



T

Node	Known	Dist	Path
s1	0	0	0
s2	0	∞	0
s3	0	∞	0
s4	0	∞	0
s5	0	∞	0

How do we get out the full paths from S to each node?

DS.GR.29

Complexity of Unweighted Shortest Path Algorithm

Like topological sort, when using adjacency lists,

Each node goes on the queue and comes off once. (How is this controlled?)

Each edge is processed once. (When?)

This leads to complexity of O(|V| + |E|).

DS.GR.30

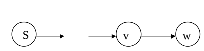
How would you go about proving correctness?

Theorem: Let w be a node at shortest distance K from node S. Then w is put on the queue at step K with its dist set to K.

Basis: K = 0

Inductive Hypothesis:
Suppose the theorem is true for distance k.

Induction: Let w be at distance k+1.
Let v be the node before w in the shortest path from S to w.



shortest distance to v is ???

2. Dijkstra's Algorithm for weighted digraphs with nonnegative weights

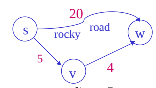
This algorithm uses the same table structure as the unweighted shortest path algorithm, but it has to do more work.

- This time we keep track of which vertices have been processed.
  - **unknown** (has not yet been selected for processing
  - **known** (has been selected and its adjacent neighbors have been updated)
- A node's Dist and Path values can be updated any time a shorter path to it is found, which can occur at any iteration, up to and including the last node processed.

The Updating Idea


```
if (v.dist + cvw < w.dist)
  {
  w.dist = v.dist + cvw;
  w.path = v;
  }
```

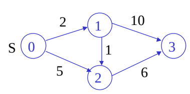
Example:



w.dist = 20
w.path = s
update node w

v.dist = 5
v.path = s

Dijkstra Example



Node	Known	Dist	Path
0	0	0	-1
1	0	∞	-1
2	0	∞	-1
3	0	∞	-1

v ← 0; 0 becomes known;
1 and 2 are adjacent to v and both unknown.
w ← 1; 0 + 2 < ∞; update 1.dist to 2 and 1.path to 0
w ← 2; 0 + 5 < ∞; update 2.dist to 5 and 2.path to 0
v ← 1 (WHY?); 1 becomes known;
2 and 3 are adjacent to 1 and both unknown.

Correctness: can be proved if there are no negative weights
Complexity: (read it)

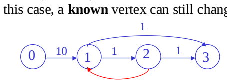
The complexity depends on how you find the unknown node with the **smallest dist**.

- If you search through all the nodes, you get
  $O(|E| + |V|^2) = O(|V|^2)$ OK for **dense** graphs
  each edge once for updates — searching |V| times
- If you keep a priority queue of nodes according to their dists, you can get down to
  O(|E| log |V|) Good for **sparse** graphs when **|E| << |V|** ^{2}

Negative Weights

What if there **are** negative weights?

The Dijkstra algorithm **fails**, because in this case, a **known** vertex can still change.



Consider the path from 0 to 3 given by

0 to 1 to 2 to 1 to 3
10 + 1 + -5 + 1 = 7

There's a problem here with both the negative weight and the loop!

The algorithm in the text works **if** there are no negative cost cycles.

Network Flow Problems

Given a weighted directed graph whose weights represent edge capacities in a flow network where:

- Through any edge (v,w), at most cvw units of "flow" may pass.
- At any vertex v that is not s or t, the total flow in must equal the total flow out.
- Vertex s, the source, has only outgoing flow.
- Vertex t, the sink, has only incoming flow.

Determine the maximum flow that can pass from s to t.

(No "best" algorithm is given.)

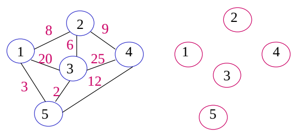
DS.GR.37

Minimal Spanning Tree

A **minimal spanning tree** (MST) of an undirected, connected, weighted graph G is a tree that includes

1. all nodes of G
2. |V| - 1 edges, connecting these nodes with no cycles

and such that the sum of the weights on these edges is smallest among all possible spanning trees.



DS.GR.38

Kruskal's Algorithm: fastest in practice

Data Structures:

- priority queue H
  - implemented as a heap
  - each entry is an edge of G
  - ordered by the edge weights
  - DeleteMin removes the smallest edge

2. union-find structure S
  - set of trees, each representing a connected component of the MST being built
  - initialized to single-node trees, one per vertex of G
  - an edge (u,v) can only be added to the MST if u and v are in separate component trees
  - when (u,v) is added to the MST, their sets are unioned in S

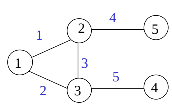
DS.GR.39

Idea of the algorithm:

Select and remove the edge with smallest weight from H.

Add it to MST if it does NOT cause a cycle, which will be if the two nodes defining the edge are in separate equivalence classes.

If you added the edge, then union the 2 classes.



What happens when it selects (1,2), then (1,3), and then (2,3)?

DS.GR.40

Worst-Case Complexity:

$$O(\underbrace{|E|}_{\text{choosing edge}}\ \underbrace{\log |E|}_{\text{heap operations}})$$

Application: Image Segmentation

- Each node represents a small square block.
- An edge contains two adjacent blocks.
- The weight on an edge is the gray-tone distance between its two blocks.

0	0	50	51
0	1	51	52
•	301	50	52
300	300	51	51

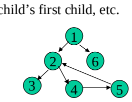
Make a node for each of the 16 blocks. Connect each adjacent pair with the distance between their given gray-tone values.

DS.GR.41

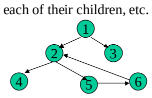
Depth-First Search and Breadth-First Search (of a directed graph)

Search a graph in a particular node order:

- depth first search
  visit a node, then its first child, then its first child's first child, etc.



- breadth-first search
  visit a node, then each of its children, then each of their children, etc.



DS.GR.42

Depth-first search can be done recursively or with a stack.

Breadth-first search uses a queue.

```
procedure BreadthFirstSearch {
  for K = 1 to NumberOfNodes
     Visited[K] = false;
  Enqueue(Start,Q);
  Visited[Start] = true;

  while (¬ isempty(Q))
     {
     V = Dequeue(Q);
     process(V);
     for each node W adjacent to V
         if (¬ Visited[W])
            {Enqueue(W,Q); Visited[W] = true;}
      }
}
```

NP-Completeness

Most problems we have studied have a polynomial complexity algorithm.

This includes both the algorithms whose complexity IS a polynomial such as

$O(N^3)$

and algorithms whose complexity can be bounded by a polynomial, such as

$O(|E||V|\log(|V|^2/|E|))$

A few algorithms we have studied have worse complexity than any polynomial.

Which algorithms are these? What complexity?

Undecidability

Another class of problems is those that are so hard that they are impossible to solve with finite resources. This is the class of undecidable problems.

The halting problem is the classical example of an undecidable problem. The problem is to design a program that can check any program (including itself) to determine if it will halt in a finite amount of time.

Let LOOP be such a program designed so that
LOOP(P) halts and prints YES if P(P) does not halt.
LOOP(P) goes into an infinite loop if P(P) halts.

Now run LOOP(LOOP). It will then
halt and print yes if LOOP(LOOP) does not halt
or go into an infinite loop if LOOP(LOOP) halts.

Since this is a contradiction, LOOP cannot exist.

The Class NP

There are problems that are in-between polynomial and unsolvable.

P is the class of polynomial-time problems.

NP is the class of nondeterministic polynomial time problems.

What is nondeterministic polynomial time?

It is the time that a procedure would take to execute on a nondeterministic machine,

that is, a machine that when it comes to a state where it must make a choice, can try all alternatives in parallel.

Where can I buy this kind of machine?

The Satisfiability Problem

The satisfiability problem takes as input a boolean expression B over some set of N boolean variables.

The problem is to determine an assignment of values to the variables to make B true.

Example: $B = (v1 \vee v2) \wedge v3$

There are 3 variables and 2^3 possible assignments.

v1	F	F	F	F	T	T	T	T
v2	F	F	T	T	F	F	T	T
v3	F	T	F	T	F	T	F	T
B	F	F	F	T	F	T	F	T

In worst case, we might try all of them, before finding a solution.

The satisfiability problem is NP-complete.

All other NP-complete problems can be reduced to any given NP-complete problem, such as the satisfiability problem.

The subgraph isomorphism problem belongs to a set of problems called consistent-labeling problems, which are NP-complete.

The problem of finding relational distance between two graphs is also NP-complete.

The problem of determining if a graph has a Hamiltonian cycle (a simple cycle that includes every vertex) is NP-complete.

The traveling salesman problem (given a complete graph with edge costs, is there a simple cycle that visits every vertex and has cost less than K) is NP-complete.

How do we solve NP-complete problems?

We try to design smart search procedures.

Instead of blindly trying every possibility in a huge search space, we try to arrange the search to prune the search space as much as possible.

Many of the techniques devised for pattern recognition and for artificial intelligence are smart searches.