playbook2017

Playbook for ICPC (Chico State ACM chapter)

Index

Environment

.vimrc Settings

```
set et sw=4 ts=4 nu
syntax on
```

Algorithms

Trees

Binary Indexed Tree

This is for sums of elements of changing list. `O(n)` setup and `O(log(n))` on each update instead of `O(n)`.

```
# Returns sum of arr[0..index]. This function assumes
# that the array is preprocessed and partial sums of
# array elements are stored in BITree[].
def getsum(BITTree,i):
    s = 0  #initialize result
    # index in BITree[] is 1 more than the index in arr[]
    i = i+1
    # Traverse ancestors of BITree[index]
    while i > 0:
        # Add current element of BITree to sum
        s += BITTree[i]
        # Move index to parent node in getSum View
        i -= i & (-i)
    return s

# Updates a node in Binary Index Tree (BITree) at given index in BITree.
# The given value 'val' is added to BITree[i] and all of its ancestors in tree.
def updatebit(BITTree , n , i ,v):
    # index in BITree[] is 1 more than the index in arr[]
    i += 1
    # Traverse all ancestors and add 'val'
    while i <= n:
        # Add 'val' to current node of BI Tree
        BITTree[i] += v
        # Update index to that of parent in update View
        i += i & (-i)

# Constructs and returns a Binary Indexed Tree for given array of size n.
def construct(arr, n):
    # Create and initialize BITree[] as 0
    BITTree = [0]*(n+1)
    # Store the actual values in BITree[] using update()
    for i in range(n):
        updatebit(BITTree, n, i, arr[i])
    # Uncomment below lines to see contents of BITree[]
    #for i in range(1,n+1):
    #      print BITTree[i],
    return BITTree
```

Lowest Common Ancestor

Finds the lowest common ancestor of two nodes in a tree.

```
# A binary tree node
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

# This function returns pointer to LCA of two given
# values n1 and n2
# This function assumes that n1 and n2 are present in
# Binary Tree
def findLCA(root, n1, n2):

    # Base Case
    if root is None:
        return None

    # If either n1 or n2 matches with root's key, report
    #  the presence by returning root (Note that if a key is
    #  ancestor of other, then the ancestor key becomes LCA
    if root.key == n1 or root.key == n2:
        return root

    # Look for keys in left and right subtrees
    left_lca = findLCA(root.left, n1, n2)
    right_lca = findLCA(root.right, n1, n2)

    # If both of the above calls return Non-NULL, then one key
    # is present in once subtree and other is present in other,
    # So this node is the LCA
    if left_lca and right_lca:
        return root

    # Otherwise check if left subtree or right subtree is LCA
    return left_lca if left_lca is not None else right_lca
```

Graphs

Bellman Ford

Single source shortest paths with negative weights.

```
import os
from numpy import *
from time import time
import sys

start_time = time()

file = open(os.path.dirname(os.path.realpath(__file__)) + "/" + sys.argv[1:][0])
vertices, edges = map(lambda x: int(x), file.readline().replace("\n", "").split(" "))

adjacency_matrix = zeros((vertices, vertices))
adjacency_matrix[:] = float("inf")
for line in file.readlines():
    tail, head, weight = line.split(" ")
    adjacency_matrix[int(head)-1][int(tail)-1] = int(weight)
```

```
def initialise_cache(vertices, s):
    cache = empty(vertices)
    cache[:] = float("inf")
    cache[s] = 0
    return cache

shortest_paths = []

for s in range(0, vertices):
    cache = initialise_cache(vertices, s)
    for i in range(1, vertices):
        previous_cache = cache[:]

        combined = (previous_cache.T + adjacency_matrix).min(axis=1)
        cache = minimum(previous_cache, combined)

        if(alltrue(cache == previous_cache)):
            break;

    # checking for negative cycles
    previous_cache = cache[:]
    combined = (previous_cache.T + adjacency_matrix).min(axis=1)
    cache = minimum(previous_cache, combined)

    print("s: " + str(s) + " done " + str(time() - start_time))
    if(not alltrue(cache == previous_cache)):
        raise Exception("negative cycle detected")

    shortest_paths.append([s, cache])

all_shortest = reduce(lambda x, y: concatenate((x,y), axis=1), map(lambda x: x[1], shortest_paths))
print(min(all_shortest))
```

Dijkstra's

Single source shortest path with no negative edges.

```
def dijkstra(graph,src,dest,visited=[],distances={},predecessors={}):
    """ calculates a shortest path tree routed in src
    """
    # a few sanity checks
    if src not in graph:
        raise TypeError('the root of the shortest path tree cannot be found in the graph')
    if dest not in graph:
        raise TypeError('the target of the shortest path cannot be found in the graph')
    # ending condition
    if src == dest:
        # We build the shortest path and display it
        path=[]
        pred=dest
        while pred != None:
            path.append(pred)
            pred=predecessors.get(pred,None)
        print('shortest path: '+str(path)+" cost="+str(distances[dest]))
    else :
        # if it is the initial  run, initializes the cost
        if not visited:
            distances[src]=0
        # visit the neighbors
        for neighbor in graph[src] :
            if neighbor not in visited:
                new_distance = distances[src] + graph[src][neighbor]
                if new_distance < distances.get(neighbor,float('inf')):
                    distances[neighbor] = new_distance
                    predecessors[neighbor] = src
        # mark as visited
        visited.append(src)
        # now that all neighbors have been visited: recurse
        # select the non visited node with lowest distance 'x'
        # run Dijskstra with src='x'
        unvisited={}
        for k in graph:
            if k not in visited:
                unvisited[k] = distances.get(k,float('inf'))
        x=min(unvisited, key=unvisited.get)
        dijkstra(graph,x,dest,visited,distances,predecessors)

if __name__ == "__main__":
    #import sys;sys.argv = ['', 'Test.testName']
    #unittest.main()
    graph = {'s': {'a': 2, 'b': 1},
            'a': {'s': 3, 'b': 4, 'c':8},
            'b': {'s': 4, 'a': 2, 'd': 2},
            'c': {'a': 2, 'd': 7, 't': 4},
            'd': {'b': 1, 'c': 11, 't': 5},
            't': {'c': 3, 'd': 5}}
    dijkstra(graph,'s','t')
```

Dijkstra's w/ Priority Queue

A faster implementation of Dijkstra's with a priority queue (heapq)

```
import heapq


class PriorityQueue(object):
    """Priority queue based on heap, capable of inserting a new node with
    desired priority, updating the priority of an existing node and deleting
    an abitrary node while keeping invariant"""

    def __init__(self, heap=[]):
        """if 'heap' is not empty, make sure it's heapified"""

        heapq.heapify(heap)
        self.heap = heap
        self.entry_finder = dict({i[-1]: i for i in heap})
        self.REMOVED = '<remove_marker>'
    def insert(self, node, priority=0):
        """'entry_finder' bookkeeps all valid entries, which are bonded in
        'heap'. Changing an entry in either leads to changes in both."""

        if node in self.entry_finder:
            self.delete(node)
        entry = [priority, node]
        self.entry_finder[node] = entry
        heapq.heappush(self.heap, entry)

    def delete(self, node):
        """Instead of breaking invariant by direct removal of an entry, mark
        the entry as "REMOVED" in 'heap' and remove it from 'entry_finder'.
        Logic in 'pop()' properly takes care of the deleted nodes."""

        entry = self.entry_finder.pop(node)
        entry[-1] = self.REMOVED
        return entry[0]

    def pop(self):
        """Any popped node marked by "REMOVED" does not return, the deleted
        nodes might be popped or still in heap, either case is fine."""

        while self.heap:
            priority, node = heapq.heappop(self.heap)
            if node is not self.REMOVED:
                del self.entry_finder[node]
                return priority, node
        raise KeyError('pop from an empty priority queue')


def dijkstra(source, pq, edges):
    """Returns the shortest paths from the source to all other nodes.
    'edges' are in form of {head: [(tail, edge_dist), ...]}, contain all
    edges of the graph, both directions if undirected."""

    size = len(pq.heap) + 1
    processed =
    uncharted = set([i[1] for i in pq.heap])
    shortest_path = {}
    shortest_path = 0
    while size > len(processed):
        min_dist, new_node = pq.pop()
        processed.append(new_node)
        uncharted.remove(new_node)
        shortest_path[new_node] = min_dist
        for head, edge_dist in edges[new_node]:
            if head in uncharted:
                old_dist = pq.delete(head)
                new_dist = min(old_dist, min_dist + edge_dist)
                pq.insert(head, new_dist)
    return shortest_path
```

Kruskal

Calculate minimum spanning-forest and its weight.

```
parent = dict()
rank = dict()

def make_set(vertice):
    parent[vertice] = vertice
    rank[vertice] = 0

def find(vertice):
    if parent[vertice] != vertice:
        parent[vertice] = find(parent[vertice])
    return parent[vertice]

def union(vertice1, vertice2):
    root1 = find(vertice1)
    root2 = find(vertice2)
    if root1 != root2:
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        else:
            parent[root1] = root2
            if rank[root1] == rank[root2]: rank[root2] += 1

def kruskal(graph):
    for vertice in graph['vertices']:
        make_set(vertice)

    minimum_spanning_tree = set()
    edges = list(graph['edges'])
    edges.sort()
    for edge in edges:
        weight, vertice1, vertice2 = edge
        if find(vertice1) != find(vertice2):
            union(vertice1, vertice2)
            minimum_spanning_tree.add(edge)
    return minimum_spanning_tree

graph = {
        'vertices': ['A', 'B', 'C', 'D', 'E', 'F'],
        'edges': set([
            (1, 'A', 'B'),
            (5, 'A', 'C'),
            (3, 'A', 'D'),
            (4, 'B', 'C'),
            (2, 'B', 'D'),
            (1, 'C', 'D'),
            ])
        }
minimum_spanning_tree = set([
            (1, 'A', 'B'),
            (2, 'B', 'D'),
            (1, 'C', 'D'),
            ])
assert kruskal(graph) == minimum_spanning_tree
```

Max Bipartite

Performs max bipartite matching (prevent nodes from sharing same destination).

```
lass Graph:
    def __init__(self,graph):
        self.graph = graph # residual graph
        self.ppl = len(graph)
        self.jobs = len(graph[0])

    # A DFS based recursive function that returns true if a
    # matching for vertex u is possible
    def bpm(self, u, matchR, seen):

        # Try every job one by one
        for v in range(self.jobs):

            # If applicant u is interested in job v and v is
            # not seen
            if self.graph[u][v] and seen[v] == False:
                seen[v] = True # Mark v as visited

                '''If job 'v' is not assigned to an applicant OR
                previously assigned applicant for job v (which is matchR[v])
                has an alternate job available.
                Since v is marked as visited in the above line, matchR[v]
                in the following recursive call will not get job 'v' again'''
                if matchR[v] == -1 or self.bpm(matchR[v], matchR, seen):
                    matchR[v] = u
                    return True
        return False

    # Returns maximum number of matching
    def maxBPM(self):
        '''An array to keep track of the applicants assigned to
        jobs. The value of matchR[i] is the applicant number
        assigned to job i, the value -1 indicates nobody is
        assigned.'''
        matchR = [-1] * self.jobs
        result = 0 # Count of jobs assigned to applicants
        for i in range(self.ppl):
            # Mark all jobs as not seen for next applicant.
            seen = [False] * self.jobs
            # Find if the applicant 'u' can get a job
            if self.bpm(i, matchR, seen):
                result += 1
        return result


bpGraph =[[0, 1, 1, 0, 0, 0],
        [1, 0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0, 0],
        [0, 0, 1, 1, 0, 0],
        [0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 1]]

g = Graph(bpGraph)

print ("Maximum number of applicants that can get job is %d " % g.maxBPM())
```

Dinic's Blocking Flow

Compute maximum flow from source to target node in a graph.

```
from queue import Queue

def dinic(graph, cap, s,t):
    """ Find maximum flow from s to t.
    returns value and matrix of flow.
    graph is list of adjacency lists, G[u]=neighbors of u
    cap is the capacity matrix
    """
    assert s!=t
    q = queue()
    #                                          -- start with empty flow
    total = 0
    flow = [[0 for _ in graph] for _ in graph]
    while True:                                 # repeat until no augment poss.
        q.put(s)
        lev = [-1]*n                            # construct levels, -1=unreach.
        lev[s] = 0
        while not q.empty():
            u = q.get()
            for v in graph[u]:
                if lev[v]==-1 and cap[u][v] > flow[u][v]:
                    lev[v]=lev[u]+1
                    q.put(v)

        if lev[t]==-1:                          # stop if target not reachable
            return (total, flow)
        upperBound = sum([cap[s][v] for v in graph[s]])
        total += dinicStep(graph, lev, cap, flow, s,t,upperBound)

def dinicStep(graph, lev, cap, flow, u,t, limit):
    """ tries to push at most limit flow from u to t. Returns how much could
    be pushed.
    """
    if limit<=0:
        return 0
    if u==t:
        return limit
    val=0
    for v in graph[u]:
        res = cap[u][v] - flow[u][v]
        if lev[v]==lev[u]+1 and res>0:
            av = dinic(graph,lev, cap,flow, v,t, min(limit-val, res))
            flow[u][v] += av
            flow[v][u] -= av
            val += av
    if val==0:
        lev[u]=-1
    return val
```

Min Cost Matching

Minimum cost bipartite matching via shortest augmenting paths.

```
import sys
import math

def MinCostMatching(cost):
    n = len(cost)
    u = [0]*n
    v = [0]*n

    for i in range(0, n):
        u[i] = min(cost[i])
    for i in range(0, n):
        v[i] = cost[0][i] - u[0]
        for j in range(0, n):
            v[i] = min(cost[j][i] - u[j], v[i])

    lmate = [-1]*n
    rmate = [-1]*n
    mated = 0;
    for i in range(0, n):
        for j in range(0, n):
            if rmate[j] == -1:
                if abs(cost[i][j] - u[i] - v[j]) < sys.float_info.epsilon:
                    lmate[i] = j
                    rmate[j] = i
                    mated += 1
                    break

    dist = [0]*n
    dad = []
    seen = []

    s = 0
    while mated < n:
        while lmate[s] != -1:
            s += 1

        dad = [-1]*n
        seen = [False]*n

        for k in range(0, n):
            dist[k] = cost[s][k] - u[s] - v[k]
        j = 0
        while True:
            j = -1
            for k in range(0, n):
                if not seen[k]:
                    if j == -1 or dist[k] < dist[j]:
                        j = k
            seen[j] = True

            if rmate[j] == -1:
                break

            i = rmate[j]
            for k in range(0, n):
                if not seen[k]:
                    new_dist = dist[j] + cost[i][k] - u[i] - v[k]
                    if dist[k] > new_dist:
                        dist[k] = new_dist
                        dad[k] = j

        for k in range(0, n):
            if not(k == j or not seen[k]):
                i = rmate[k]
                v[k] += dist[k] - dist[j]
                u[i] -= dist[k] - dist[j]
        u[s] += dist[j]

        while dad[j] >= 0:
            d = dad[j]
            rmate[j] = rmate[d]
            lmate[rmate[j]] = j
            j = d

        rmate[j] = s
        lmate[s] = j
        mated += 1

    value = 0
    print (rmate, lmate)
    for i in range(0, n):
        print(cost[i][lmate[i]])
        value += cost[i][lmate[i]]
    return value
```

Min Cost Max Flow

Computes the minimum cost max flow on a graph with the Edmonds and Karp algorithm.

```
import decimal

def EdmondsKarp(E, C, s, t):
    n = len(C)
    flow = 0
    F = [[0 for y in range(n)] for x in range(n)]
    while True:
        P = [-1 for x in range(n)]
        P[s] = -2
        M = [0 for x in range(n)]
        M[s] = decimal.Decimal('Infinity')
        BFSq = []
        BFSq.append(s)
        pathFlow, P = BFSEK(E, C, s, t, F, P, M, BFSq)
        if pathFlow == 0:
            break
        flow = flow + pathFlow
        v = t
        while v != s:
            u = P[v]
            F[u][v] = F[u][v] + pathFlow
            F[v][u] = F[v][u] - pathFlow
            v = u
    return flow

def BFSEK(E, C, s, t, F, P, M, BFSq):
    while (len(BFSq) > 0):
        u = BFSq.pop(0)
        for v in E[u]:
            if C[u][v] - F[u][v] > 0 and P[v] == -1:
                P[v] = u
                M[v] = min(M[u], C[u][v] - F[u][v])
                if v != t:
                    BFSq.append(v)
                else:
                    return M[t], P
    return 0, P
```

Min Cut

Computes the minimum edge cut on a given graph based on the Stoer Wagner algorithm.

```
import sys

def sum_into_A(G, A, u):
    total = 0
    for v in G[u].keys():
        if v in A:
            total += G[u][v]

    return total

def merge(G, u, v):
    # Merge v into u
    for q in G[v].keys():
        if q == u:
            # Remove existing edge from u to v
            del G[u][v]
        elif q in G[u].keys():
            # Combine common edge between v and u
            G[u][q] += G[v][q]
            G[q][u] = G[u][q]

            # Remove reference to v
            if v in G[q]:
                del G[q][v]
        else:
            # Add v's edge to u
            G[u][q] = G[v][q]

            # Redirect q's edge to v to u
            del G[q][v]
            G[q][u] = G[v][q]

    # Remove v from G
    del G[v]

def minimum_cut_phase(G, u):
    # Maintain mutually-exclusive sets of vertices whose union is G.V
    A = [u]
    not_A = []
    G_len = 0
    for v in G.keys():
        G_len += 1
        if v != u:
            not_A.append(v)

    # So we don't have to call len() during loop
    not_A_len = G_len - 1

    # Keep adding most tightly-connected vertex to A
    mtc = u
    prev_mtc = not_A[0]
    while not_A_len > 1:
        cur_max = -1
        for u in not_A:
            u_sum = sum_into_A(G, A, u)
            if u_sum > cur_max:
                cur_max = u_sum
                mtc = u

        A.append(mtc)
        not_A.remove(mtc)
        not_A_len -= 1

        prev_mtc = mtc

    mtc = not_A[0]
    cut_of_phase = sum_into_A(G, A, mtc)

    # Merge mtc and prev_mtc
    merge(G, mtc, prev_mtc)

    return cut_of_phase
```

```
def stoer_wagner(G):
    min_cut = sys.maxsize
    while len(G) > 1:
        cut_of_phase = minimum_cut_phase(G, 0)
        min_cut = min(min_cut, cut_of_phase)

    return min_cut
```

Prim

This builds a minimum weight spanning tree for a given graph.

```
from pythonds.graphs import PriorityQueue, Graph, Vertex

def prim(G,start):
    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)
```

Eulerian Path

Path that visits every edge exactly once.

```
def eulerPath(graph):
    # counting the number of vertices with odd degree
    odd = [ x for x in graph.keys() if len(graph[x])&1 ]
    odd.append( graph.keys()[0] )

    if len(odd)>3:
        return None

    stack = [ odd[0] ]
    path = []

    # main algorithm
    while stack:
        v = stack[-1]
        if graph[v]:
            u = graph[v][0]
            stack.append(u)
            # deleting edge u-v
            del graph[u][ graph[u].index(v) ]
            del graph[v][0]
        else:
            path.append( stack.pop() )

    return path
```

Strongly Connected Components

Computes a graph of strongly connected components

```
def strongly_connected_components_path(vertices, edges):
    identified = set()
    stack = []
    index = {}
    boundaries = []

    def dfs(v):
        index[v] = len(stack)
        stack.append(v)
        boundaries.append(index[v])

        for w in edges[v]:
            if w not in index:
                # For Python >= 3.3, replace with "yield from dfs(w)"
                for scc in dfs(w):
                    yield scc
            elif w not in identified:
                while index[w] < boundaries[-1]:
                    boundaries.pop()

        if boundaries[-1] == index[v]:
            boundaries.pop()
            scc = set(stack[index[v]:])
            del stack[index[v]:]
            identified.update(scc)
            yield scc

    for v in vertices:
        if v not in index:
            # For Python >= 3.3, replace with "yield from dfs(v)"
            for scc in dfs(v):
                yield scc


def strongly_connected_components_tree(vertices, edges):
    identified = set()
    stack = []
    index = {}
    lowlink = {}

    def dfs(v):
        index[v] = len(stack)
        stack.append(v)
        lowlink[v] = index[v]
        for w in edges[v]:
            if w not in index:
                # For Python >= 3.3, replace with "yield from dfs(w)"
                for scc in dfs(w):
                    yield scc
                lowlink[v] = min(lowlink[v], lowlink[w])
            elif w not in identified:
                lowlink[v] = min(lowlink[v], lowlink[w])

        if lowlink[v] == index[v]:
            scc = set(stack[index[v]:])
            del stack[index[v]:]
            identified.update(scc)
            yield scc

    for v in vertices:
        if v not in index:
            # For Python >= 3.3, replace with "yield from dfs(v)"
            for scc in dfs(v):
                yield scc


def strongly_connected_components_iterative(vertices, edges):
    identified = set()
    stack = []
    index = {}
    boundaries = []

    for v in vertices:
        if v not in index:
            to_do = [('VISIT', v)]
            while to_do:
                operation_type, v = to_do.pop()
                if operation_type == 'VISIT':
                    index[v] = len(stack)
                    stack.append(v)
                    boundaries.append(index[v])
                    to_do.append(('POSTVISIT', v))
                    # We reverse to keep the search order identical to that of
                    # the recursive code;  the reversal is not necessary for
                    # correctness, and can be omitted.
                    to_do.extend(
                        reversed([('VISITEDGE', w) for w in edges[v]]))
                elif operation_type == 'VISITEDGE':
                    if v not in index:
                        to_do.append(('VISIT', v))
                    elif v not in identified:
                        while index[v] < boundaries[-1]:
                            boundaries.pop()
                else:
                    # operation_type == 'POSTVISIT'
                    if boundaries[-1] == index[v]:
                        boundaries.pop()
                        scc = set(stack[index[v]:])
                        del stack[index[v]:]
                        identified.update(scc)
                        yield scc
```

Topological Sort

This topoligically sorts nodes from a graph.

```
def topolgical_sort(graph_unsorted):
    # This is the list we'll return, that stores each node/edges pair
    # in topological order.
    graph_sorted = []

    # Convert the unsorted graph into a hash table. This gives us
    # constant-time lookup for checking if edges are unresolved, and
    # for removing nodes from the unsorted graph.
    graph_unsorted = dict(graph_unsorted)

    # Run until the unsorted graph is empty.
    while graph_unsorted:
        acyclic = False
        for node, edges in list(graph_unsorted.items()):
            for edge in edges:
                if edge in graph_unsorted:
                    break
            else:
                acyclic = True
                del graph_unsorted[node]
                graph_sorted.append((node, edges))

        if not acyclic:
            # Uh oh, we've passed through all the unsorted nodes and
            # weren't able to resolve any of them, which means there
            # are nodes with cyclic edges that will never be resolved,
            # so we bail out with an error.
            raise RuntimeError("A cyclic dependency occurred")

    return graph_sorted
```

Floyd-Warshall

Computes an all-pairs shortest path on a given graph.

```
def floydwarshall(graph):

    # Initialize dist and pred:
    # copy graph into dist, but add infinite where there is
    # no edge, and 0 in the diagonal
    dist = {}
    pred = {}
    for u in graph:
        dist[u] = {}
        pred[u] = {}
        for v in graph:
            dist[u][v] = 1000
            pred[u][v] = -1
        dist[u][u] = 0
        for neighbor in graph[u]:
            dist[u][neighbor] = graph[u][neighbor]
            pred[u][neighbor] = u

    for t in graph:
        # given dist u to v, check if path u - t - v is shorter
        for u in graph:
            for v in graph:
                newdist = dist[u][t] + dist[t][v]
                if newdist < dist[u][v]:
                    dist[u][v] = newdist
                    pred[u][v] = pred[t][v] # route new path through t

    return dist, pred



graph = {0 : {1:6, 2:8},
         1 : {4:11},
         2 : {3: 9},
         3 : {},
         4 : {5:3},
         5 : {2: 7, 3:4}}

dist, pred = floydwarshall(graph)
print "Predecesors in shortest path:"
for v in pred: print "%s: %s" % (v, pred[v])
print "Shortest distance from each vertex:"
for v in dist: print "%s: %s" % (v, dist[v])
```

Geometry

Convex Hull

Computes the positions of points for the convex hull of a set of given points. The polygon drawing the smallest area containing given points.

```
def convex_hull(points):
    """Computes the convex hull of a set of 2D points.

    Input: an iterable sequence of (x, y) pairs representing the points.
    Output: a list of vertices of the convex hull in counter-clockwise order,
      starting from the vertex with the lexicographically smallest coordinates.
    Implements Andrew's monotone chain algorithm. O(n log n) complexity.
    """

    # Sort the points lexicographically (tuples are compared lexicographically).
    # Remove duplicates to detect the case we have just one unique point.
    points = sorted(set(points))

    # Boring case: no points or a single point, possibly repeated multiple times.
    if len(points) <= 1:
        return points

    # 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.
    # Returns a positive value, if OAB makes a counter-clockwise turn,
    # negative for clockwise turn, and zero if the points are collinear.
    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    # Build lower hull
    lower = []
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)

    # Build upper hull
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)

    # Concatenation of the lower and upper hulls gives the convex hull.
    # Last point of each list is omitted because it is repeated at the beginning of the other list.
    return lower[:-1] + upper[:-1]


# Example: convex hull of a 10-by-10 grid.
assert convex_hull([(i//10, i%10) for i in range(100)]) == [(0, 0), (9, 0), (9, 9), (0, 9)]
```

Fast Fourier Transform

```
import math

def complex_dft(xr, xi, n):
    pi = 3.141592653589793
    rex = [0] * n
    imx = [0] * n
    for k in range(0, n):  # exclude n
        rex[k] = 0
        imx[k] = 0
    for k in range(0, n):  # for each value in freq domain
        for i in range(0, n):  # correlate with the complex sinusoid
            sr =  math.cos(2 * pi * k * i / n)
            si = -math.sin(2 * pi * k * i / n)
            rex[k] += xr[i] * sr - xi[i] * si
            imx[k] += xr[i] * si + xi[i] * sr
    return rex, imx

# FFT version based on the original BASIC program
def fft_basic(rex, imx, n):
    pi = 3.141592653589793
    m = int(math.log(n, 2))  # float to int
    j = n / 2
    # bit reversal sorting
    for i in range(1, n - 1):  # [1,n-2]
        if i >= j:
            # swap i with j
            print "swap %d with %d"%(i, j)
            rex[i], rex[j] = rex[j], rex[i]
            imx[i], imx[j] = imx[j], imx[i]
        k = n / 2
        while (1):
            if k > j:
                break
            j -= k
            k /= 2
        j += k

    for l in range(1, m + 1):  # each stage
        le = int(math.pow(2, l))  # 2^l
        le2 = le / 2
        ur = 1
        ui = 0
        sr =  math.cos(pi / le2)
        si = -math.sin(pi / le2)
        for j in range(1, le2 + 1):  # [1, le2] sub DFT
            for i in xrange(j - 1, n - 1, le):  #  for butterfly
                ip = i + le2
                tr = rex[ip] * ur - imx[ip] * ui
                ti = rex[ip] * ui + imx[ip] * ur
                rex[ip] = rex[i] - tr
                imx[ip] = imx[i] - ti
                rex[i] += tr
                imx[i] += ti
            tr = ur
            ur = tr * sr - ui * si
            ui = tr * si + ui * sr

def print_list(l):
    n = len(l)
    print "[%d]: {"%(n)
    for i in xrange(0, n):
        print l[i],
    print "}"


if __name__ == "__main__":
    print "hello,world."
    pi = 3.1415926
    x = []
    n = 64
    for i in range(0, n):
        p = math.sin(2 * pi * i / n)
        x.append(p)

    xr = x[:]
    xi = x[:]
    rex, imx = complex_dft(xr, xi, n)
    print "complet_dft(): n=", n
    print "rex: "
    print_list([int(e) for e in rex])
    print "imx: "
    print_list([int(e) for e in imx])

    fr = x[:]
    fi = x[:]

    fft_basic(fr, fi, n)
    print "fft_basic(): n=", n
    print "rex: "
    print_list([int(e) for e in fr])
    print "imx: "
    print_list([int(e) for e in fi])
```

Math

Euclid Routines

A set of routines to do common math operations.

```
# a % b positive
def mod(a, b):
    return ((a%b) + b) % b

def gcd(a, b):
    while b:
        t = a%b
        a = b
        b = t
    return a

def lcm(a, b):
    return a // gcd(a, b) * b

# (a^b) % m via successive squaring
def powermod(a, b, m):
    ret = 1
    while b:
        if b & 1:
            ret = mod(ret * a, b)
        a = mod(a * a, m)
        b //= 2
    return ret
```

Fast Exponential

```
def power(n, k):
    ret = 1
    while k:
        if k & 1:
            ret *= x
        k //= 2
        x *= x
    return ret
```

Gauss Jordan (full pivoting)

```
def gauss_jordan(m, eps = 1.0/(10**10)):
    """Puts given matrix (2D array) into the Reduced Row Echelon Form.
         Returns True if successful, False if 'm' is singular.
         NOTE: make sure all the matrix items support fractions! Int matrix will NOT work!
         Written by Jarno Elonen in April 2005, released into Public Domain"""
    (h, w) = (len(m), len(m[0]))
    for y in range(0,h):
        maxrow = y
        for y2 in range(y+1, h): # Find max pivot
            if abs(m[y2][y]) > abs(m[maxrow][y]):
                maxrow = y2
        (m[y], m[maxrow]) = (m[maxrow], m[y])
        if abs(m[y][y]) <= eps: # Singular?
            return False
        for y2 in range(y+1, h): # Eliminate column y
            c = m[y2][y] / m[y][y]
            for x in range(y, w):
                m[y2][x] -= m[y][x] * c
    for y in range(h-1, 0-1, -1): # Backsubstitute
        c    = m[y][y]
        for y2 in range(0,y):
            for x in range(w-1, y-1, -1):
                m[y2][x] -=    m[y][x] * m[y2][y] / c
        m[y][y] /= c
        for x in range(h, w): # Normalize row y
            m[y][x] /= c
    return True
```

Primes

```
def is_prime(number):
    for i in range(2,1+int(math.sqrt(number))):
        if number % i == 0:
            return False
    return True

primes = [2]
for i in range(3,10000,2):
    if is_prime(i):
        primes.append(i)
```

String/List

KMP Search

Linear runtime complexity string search.

```
def KnuthMorrisPratt(text, pattern):

    """Yields all starting positions of copies of the pattern in the text.
Calling conventions are similar to string.find, but its arguments can be
lists or iterators, not just strings, it returns all matches, not just
the first one, and it does not need the whole text in memory at once.
Whenever it yields, it will have read the text exactly up to and including
the match that caused the yield."""

    # allow indexing into pattern and protect against change during yield
    pattern = list(pattern)

    # build table of shift amounts
    shifts = [1] * (len(pattern) + 1)
    shift = 1
    for pos in range(len(pattern)):
        while shift <= pos and pattern[pos] != pattern[pos-shift]:
            shift += shifts[pos-shift]
        shifts[pos+1] = shift

    # do the actual search
    startPos = 0
    matchLen = 0
    for c in text:
        while matchLen == len(pattern) or \
              matchLen >= 0 and pattern[matchLen] != c:
            startPos += shifts[matchLen]
            matchLen -= shifts[matchLen]
        matchLen += 1
        if matchLen == len(pattern):
            yield startPos
```

Longest Common Subsequence

```
def lcs(a, b):
    lengths = [[0 for j in range(len(b)+1)] for i in range(len(a)+1)]
    # row 0 and column 0 are initialized to 0 already
    for i, x in enumerate(a):
        for j, y in enumerate(b):
            if x == y:
                lengths[i+1][j+1] = lengths[i][j] + 1
            else:
                lengths[i+1][j+1] = max(lengths[i+1][j], lengths[i][j+1])
    # read the substring out from the matrix
    result = ""
    x, y = len(a), len(b)
    while x != 0 and y != 0:
        if lengths[x][y] == lengths[x-1][y]:
            x -= 1
        elif lengths[x][y] == lengths[x][y-1]:
            y -= 1
        else:
            assert a[x-1] == b[y-1]
            result = a[x-1] + result
            x -= 1
            y -= 1
    return result
```

Longest Increasing Subsequence

```
def longest_increasing_subsequence(X):
    """Returns the Longest Increasing Subsequence in the Given List/Array"""
    N = len(X)
    P = [0] * N
    M = [0] * (N+1)
    L = 0
    for i in range(N):
       lo = 1
       hi = L
       while lo <= hi:
           mid = (lo+hi)//2
           if (X[M[mid]] < X[i]):
               lo = mid+1
           else:
               hi = mid-1

       newL = lo
       P[i] = M[newL-1]
       M[newL] = i

       if (newL > L):
           L = newL

    S = []
    k = M[L]
    for i in range(L-1, -1, -1):
        S.append(X[k])
        k = P[k]
    return S[::-1]
```

Other

CSP

```
"""
To use this code, call csp(variables, constraints).
variables is a dictionary containing each variable and its possible values.
constraints is a tuple with (variable1, variable2, function) where the function
is expected to return either true or false based on a constraint. This means
you will need to define a function for each different kind of constraint.
csp will return a dictionary of variables with assigned values that satisfy
the csp.
"""

FAILURE = 'FAILURE'

def csp(variables, constraints):
    csp = {'variables':variables, 'constraints':constraints}
    result = solve(csp)
    return result

def solve(csp):
  """
  Solve a constraint satisfaction problem.
  csp is an object that should have properties:
    variables:
      dictionary of variables and values they can take on
    constraints:
      list of constraints where each element is a tuple of
      (head node, tail node, constraint function)
 """
  result = backtrack({}, csp['variables'], csp)
  if result == FAILURE: return result
  return { k:v[0] for k,v in result.iteritems() } # Unpack values wrapped in arrays.

def backtrack(assignments, unassigned, csp):
  """
  Main algorithm for solving a constraint satisfaction problem.
  """
  if finished(unassigned): return assignments
  var = select_unassigned_variable(unassigned)
  values = order_values(var, assignments, unassigned, csp)
  del unassigned[var]

  for value in values:
    assignments[var] = [value]
    v = enforce_consistency(assignments, unassigned, csp)
    if any_empty(v): continue # A variable has no legal values.
    u = { var:val for var,val in v.iteritems() if var not in assignments }
    result = backtrack(assignments.copy(), u, csp)
    if result != FAILURE: return result

  return FAILURE

def finished(unassigned):
  return len(unassigned) == 0

def any_empty(v):
  return any((len(values) == 0 for values in v.itervalues()))

def partial_assignment(assignments, unassigned):
  """
  Merge together assigned and unassigned dictionaries (assigned
  values take priority).
  """
  v = unassigned.copy()
  v.update(assignments)
  return v

def enforce_consistency(assignments, unassigned, csp):
  """
```

```python
  Enforces arc consistency by removing values from tail nodes of a
  constraint, and if a node loses value, perform arc consistency on
  that node.
  """

  def remove_inconsistent_values(head, tail, constraint, variables):
    """
    Checks if there are any inconsistent values in the tail. An
    inconsistent value means that for a given value in the tail,
    there are no values in head that will satisfy the constraints.
    Returns whether there were inconsistent values in tail.
    """
    valid_tail_values = [t for t in variables[tail] if any((constraint(h, t) for h in variables[head]))
    removed = len(variables[tail]) != len(valid_tail_values)
    variables[tail] = valid_tail_values
    return removed

  def incoming_constraints(node):
    """
    All constraints where constraint head is the passed in node.
    """
    return [(h, t, c) for h, t, c in csp['constraints'] if h == node]

  queue, variables = csp['constraints'][:], partial_assignment(assignments, unassigned)
  while len(queue):
    head, tail, constraint = queue.pop(0)
    if remove_inconsistent_values(head, tail, constraint, variables):
      queue.extend(incoming_constraints(tail)) # Need to recheck constraint arcs coming into tail.
  return variables

def select_unassigned_variable(unassigned):
  """
  Picks the next variable to assign according to the
  Minimum Remaining Values principle: choose the variable
  with the fewest legal values remaining. This helps
  identify failure earlier.
  """
  return min(unassigned.keys(), key=lambda k: len(unassigned[k]))

def order_values(var, assignments, unassigned, csp):
  """
  Orders the values of an unassigned variable according to the
  Least Constraining Value principle: order values by the amount
  of values they eliminate when assigned (fewest eliminated at the
  front, most eliminated at the end). Keeps future options open.
  """
  def count_vals(vars):
    return sum((len(vars[v]) for v in unassigned if v != var))

  def values_eliminated(val):
    assignments[var] = [val]
    new_vals = count_vals(enforce_consistency(assignments, unassigned, csp))
    del assignments[var]
    return new_vals
  return sorted(unassigned[var], key=values_eliminated, reverse=True)
```

Cheatsheet

Reading Input

Reading a pair of space separated values `n` and `k`:

```
(n,k) = [int(x) for x in input().split()]
```

Reading a list of values

```
list = [int(x) for x in input().split()]
```