Graphs And Algorithms

1 Graphics Basics

1.1 Definitions

parallel arcs two arcs with the same endpoints

loops an arc with both endpoints the same

1.1.1 Theorems

In any graph, the total degrees of all nodes is twice the number of arcs.

In any graph, the number of odd nodes is even.

2 Graph Isomorphism and Planar Graphs

2.1 Isomorphism

Graphs G and G' are **isomorphic** iff:

- Same number of nodes
- Same number of arcs
- Same number of loops
- Nodes have same degrees
- There is a bijection from G to G' to map arcs

2.2 Automorphism

An **automorphism** on G is an isomorphism from G to itself. It is **nontrivial** if it is not the identity.

2.3 Planer

A graph is **planer** if it can be embedded into a plane, that is, its arcs can be drawn as straight lines which do not cross.

Two graphs are **homeomorphic** if they can be obtained from the same graph by a series of operations replacing an arc $x - y$ by two arcs $x - z - y$.

Every simple planer graph can be a **map**.

Kuratowski's theorem A graph is planar iff it does not contain a subgraph homeomorphic to K_5 or $K_{3,3}$.

Eular's theorem Let G be a connected planar graph. Let G have N nodes, A arcs and F faces. Then $F = A - N + 2$

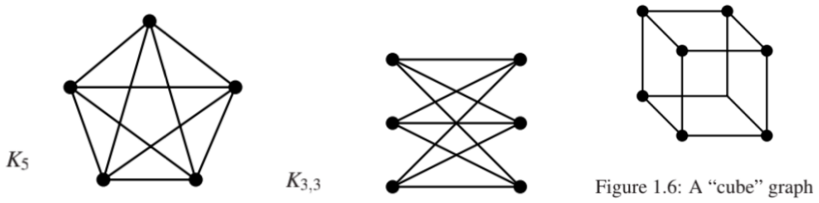

Figure 1.6: A "cube" graph

2.3.1 Bipartite

A graph G is **bipartite** if $nodes(G)$ can be partitioned into two sets X and Y in such a way that no arc of G joins any two members of X, and no arc joins any two members of Y. For example, $K_{3,3}$ and the cube.

2.3.2 Propositions

2-colourable A graph is bipartite iff it is **2-colourable**.

Four Colour Theorem Every map(simple planer graph) is 4-colourable

3 Connectedness, Euler Paths and Hamiltonian Circuits

3.1 Paths and Cycles

connected graph A graph is connected if forall different nodes n and n', there is a path from n to n'.

Euler path is a path which uses each arc exactly once.

Euler circuit (aka **Euler cycle**) is a cycle which uses each arc exactly once.

3.1.1 Propositions

A connected graph has an Euler path iff the number of odd nodes is either 0 or 2.

A connected graph has an Euler circuit iff every node has even degree.

3.2 Hamiltonian Circuits

Hamiltonian path is a path which visits every node in a graph exactly once.

Hamiltonian circuit is a cycle which visits every node exactly once.

3.2.1 Finding Hamiltonian path

The (not conclusive) conditions for a graph to have Hamiltonian path:

1. Each node must have degree at least two, since we must approach it and leave it via two different nodes.

Remove loops and parallel arcs, since they make no difference in finding Hamiltonian paths.

4 Trees

rooted graph is a pair (G, x) where G is a graph and $x \in nodes(G)$. The node x is the root of the tree.

rooted tree is a rooted, acyclic, connected graph.

non-rooted tree is an acyclic, connected graph.

tree sometimes refer to *non-rooted tree*.

depth of node is the length of the path from the root to the node.

parent is the unique node adjacent to x on the path from x to the root.

depth of tree is the maximum of depths of all its nodes.

For any rooted or non-rooted tree T with n nodes:

There is a unique path between any nodes.

There are $n - 1$ arcs.

4.1 Spanning Trees

Spanning tree An non-rooted tree T is said to be a spanning tree for G if T spans G, i.e. T is a subgraph of G and nodes(T) = nodes(G)

Any connected graph with n nodes has a spanning tree of $n - 1$ arcs.

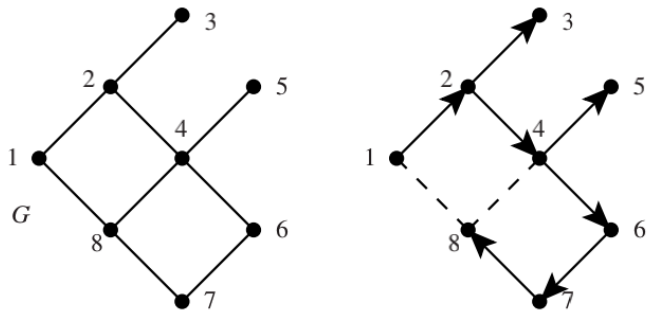

Figure 1: Depth-first Search

5 Directed Graphs

directed graph is a set N of nodes and a set A of arcs such that each $a \in A$ is associated with an ordered pair of nodes (the endpoints of a).

arc can be referred as (x, y), for arc starting from x and ending with y.

indegree is the number of arcs entering the node.

outdegree is the number of arcs leaving the node.

strongly connected A graph is **strongly connected** if for every different nodes there is an arc connecting the two nodes.

6 Graph Traversal

Depth-first Search (Figure **??**) We start from a particular node start, and we continue outwards along a path from start until we reach a node which has no adjacent unvisited nodes. We then backtrack to the previous node and try a different path.

Beadth-first Search (Figure **??**) We start from a particular node start and we fan out from there to all adjacent nodes, from which we then run breadth-first searches.

6.1 Implementing DFS and BFS

```
procedure dfs(x):
  visited[x] = true
  print x
  for y in adj[x]:
    if not visited[y]:
```

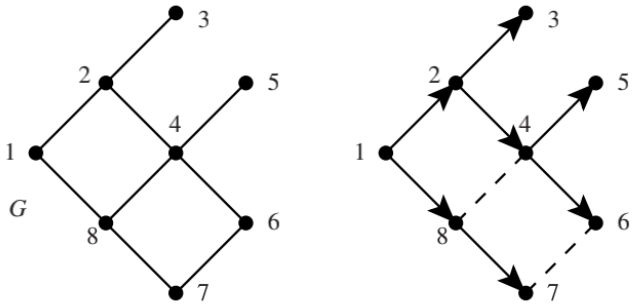

Figure 2: Breadth-first Search

```
    parent[y] = x
    dfs(y)
    # backtrack to x

procedure bfs(x) {
  visited[x] = true
  print x
  enqueue(x,Q)
  while not isempty(Q):
    y = front(Q)
    for z in adj[y]:
      if not visited[z]:
         visited[z] = true
         print z
         parent[z] = y
         enqueue(z,Q)
    dequeue(Q)
}
```

6.2 Applications

6.2.1 Determining Whether a Graph Is Connected

Apply either DFS or BFS. Complexity: $O(n + m)$

6.2.2 Determining Whether a Graph Has a Cycle

If a graph has $\geq n$ arcs then it has a cycle. We can use DFS to find the cycle.

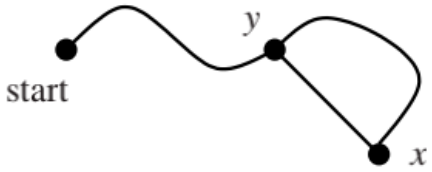

Figure 3: Using DFS to find cycles

Proposition Let G be a connected graph, and let T be a spanning tree of Gobtained by DFS starting at node start. If a is any arc of G (not necessarily in T), with endpoints x and y, then either x is an ancestor of y in T or y is an ancestor of x in T .

Here 'x is an ancestor of y in T' means that x lies on the (unique) path from start to y in T.

```
procedure cycleDfs(x):
  visited[x] = true
  # print x
  for y in adj[x]:
    if visited[y] and y ≠ parent[x]:
      # cycle found involving x and y
      return (x,y)
    if not visited[y]:
      parent[y] = x
      pair = cycleDfs(y)
      # backtrack to x
      if pair:
        return pair
```

Suppose that nodes x and y are returned. Then y must be an ancestor of x, as in Figure **??**. The cycle will be $x, \text{parent}[x], \ldots, y$.

6.2.3 Calculating Distances From the Start Node

BFS finds the shortest path from the start node to any reachable node, while DFS may well give a longer distance than necessary.

```
procedure shortestPathAndDistanceBFS(x):
  visited[x] = true
  distance[x] = 0
  enqueue(x,Q)
```

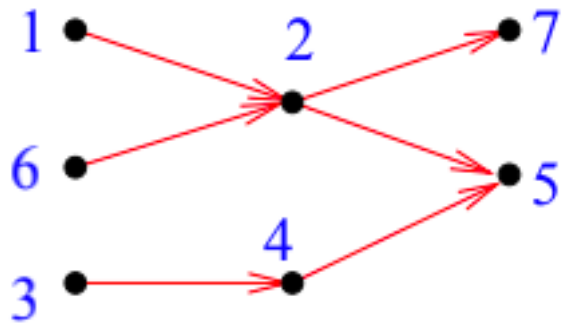

Figure 4: Topological Sorting: Directed Graph Example

```
while not isempty(Q):
  y = front(Q)
  for z in adj[y]:
    if not visited[z]:
      visited[z] = true
      parent[z] = y
      distance[z] = distance[y] + 1
      enqueue(z,Q)
  dequeue(Q)
# Now distance[x] is the shortest distance of x to the start.
```

Clearly the shortest path from a node x to the start node can be read off from the parent array as x, $parent[x]$, $parent[parent[x]]$, ..., $start$.

6.2.4 Topological Sorting

Given a directed acyclic graph (DAG) G with n nodes, find a total ordering of the nodes x_1, ..., x_n such that for any $i, j \leq n$, if $j\rangle i$ then there is no path from x_j to x_i in G. Such a total ordering is called a **topological sort** of G. It could be presented as a list or array of nodes.
For the diagram in Figure **??** a topological sort could be 1,6,3,2,4,7,5 or 6,1,2,7,3,4,5, etc.
Topological sort can be calculated by reversing the exit order of a DFS.

6.2.5 Implementation

Given: a directed graph G with n nodes. Return: topological sort of G as array ts of nodes if G acyclic (else abort).

```
procedure dfsts(x)
  entered[x] = true
```

```
  for y in adj[x]:
    if entered[y]:
      if not exited[y]:
        abort # cycle
    else:
      parent[y] = x
      dfsts(y)
  exited[x] = true
  ts[index] = x
  index = index - 1

index = n - 1
for x in nodes(G):
if not entered[x]:
dfsts(x)
```

7 Weighted Graphs

7.1 Minimal Spanning Trees

T is a **minimum spanning tree** (MST) for G if T is a spanning tree for G and no other spanning tree for G has smaller weight.

7.2 Prim's Algorithm

To construct a MST starting from a root node, at each stage, **Prim's Algorithm** adds the shortest arc to extend the tree. So Prim's algorithm is said to be greedy.
At an arbitrary stage in Prim's MST algorithm, there are three kinds of nodes:

tree nodes are the ones already in the tree

fringe nodes are candidates to join at the next stage

unseen nodes are the rest

Initially all nodes are unseen.
See Figure **??**
Informal algorithm:

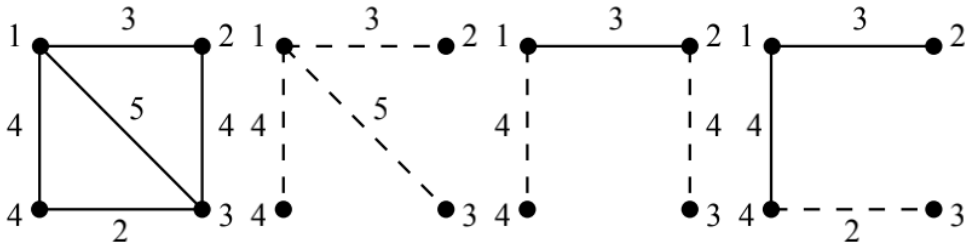

Figure 5: MST constructed by Prim's algorithm

```
Choose any node start as the root
Reclassify start as tree
Reclassify all nodes adjacent to start as fringe
while fringe nonempty:
  Select an arc of minimum weight between a tree node t and a fringe node f (*)
  Reclassify f as tree
  Add arc (t, f) to the tree
  Reclassify all unseen nodes adjacent to f as fringe
```

7.2.1 Prim's MST Algorithm classic

```
tree[start] = true
for x in adj[start]:
  # add x to fringe
  fringe[x] = true
  parent[x] = start
  weight[x] = W[start, x]
while fringe nonempty:
  Select a fringe node f such that weight[f] is minimum
  fringe[f] = false
  tree[f] = true
  for y in adj[f]:
    if not tree[y]:
      if fringe[y]:
        # update candidate arc
        if W[f, y] < weight[y]:
          weight[y] = W[f, y]
          parent[y] = f
```

```
      else:
        # y is unseen
        fringe[y] = true
        weight[y] = W[f, y]
        parent[y] = f
```

Complexity: $O(n^2)$

7.2.2 Prim's MST Algorithm with Priority Queue

```
Q = PQcreate()
for x in NodesG:
  key[x] = ∞
  parent[x] = nil
  insert(Q, x)
updateKey(Q, start, 0)
while not isEmpty(Q):
  f = getMin(Q)
  deleteMin(Q)
  tree[f] = true
  for y in adj[f]:
    if not tree[y]: # so y in Q
      if W[ f, y] < key[y]:
        updateKey(Q, y, W[f, y])
        parent[y] = f
```

Complexity compared with classic Prim:

When graph is sparse, say $m \leq n \log n$, complexity is $O(m\log n) = O(n\log^2 n)$, which is better than $O(n^2)$, so Prim with PQ is better.

When graph is dense, with $O(n^2)$ arcs, complexity is $O(m\log n) = O(n^2 \log n)$, which is worse than $O(n^2)$, so classic Prim is better.

7.3 Kruskal's Algorithm

Kruskal's Algorithm is more greedy than Prim's —it chooses the shortest arc not yet in the tree, except when this would give a cycle. See Figure **??**.

For the implementation of Kruskal's algorithm, we have to do two things:

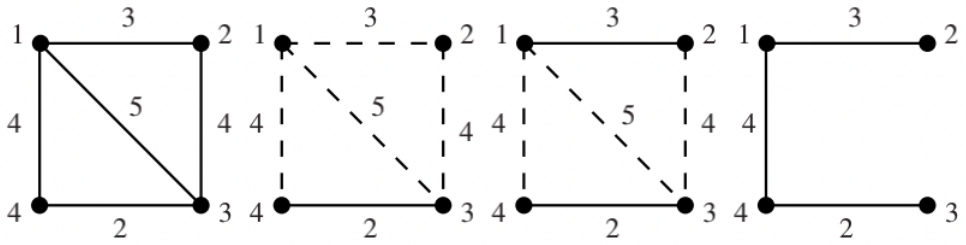

Figure 6: MST constructed by Kruskal's algorithm

1. Look at each arc in ascending order of weight. We can use a priority queue here (or just sort the arcs at the start).

2. Check whether adding the arc to the forest so far creates a cycle.
   Here we use *dynamic equivalence classes*.
   Put nodes in the same equivalence class if they belong to the same connected component of the forest constructed so far.
   An arc (x, y) can be added if x and y belong to different equivalence classes.
   If (x, y) is added, then merge the equivalence classes of x and y.

Dynamic equivalence classes can be handled using the **Union-Find** data type. Each set has a **leader** element which is the representative of that set.

find Find the leader of the equivalence class

union Merge two classes.

```
Let G have n nodes numbered from 1 to n.
Build a priority queue Q of the edges of G with the weights as keys
sets = UFcreate(n) # initialise Union-Find with singletons {1},...,{n}
F = 0/ # forest being constructed
while not isEmpty(Q):
  (x, y) = getMin(Q)
  deleteMin(Q)
  x' = find(sets, x)
  y' = find(sets, y)
  if x' != y': # no cycle
    add (x, y) to F
    union(sets, x', y') # merge the two components
```

Note: implementation of Union-find can be found in Lecture Nodes on page 36.
Overall complexity for Kruskal is $O(m \log n)$, the same as Prim with PQ.

7.3.1 Comparing complexity of Prim's and Kruskal's algorithms

- On dense graphs, where m is large (order n^2), then Kruskal gives $O(n^2 \log n)$ so classic Prim is preferred.
- On sparse graphs, then Kruskal and Prim with PQ are better, with $O(m \log n) = O(n \log^2 n)$

7.4 The Shortest Path Problem

THE SHORTEST PATH PROBLEM Given a weighted graph (G, W), and two nodes start and finish, find the shortest path from start to finish in the graph.

THE ALL PAIRS SHORTEST PATH PROBLEM Given a weighted directed graph G, find the shortest paths between *all* pairs of nodes of G.

7.5 Dijkstra's Algorithm

Dijkstra's Shortest Path Algorithm is very related to Prim's MST algorithm.
Note that Prim's algorithm forms minimal spanning tree, but Dijkstra's only forms a spanning tree, not necessarily minimal.

7.5.1 Idea

tree nodes are the ones already in the tree *(similar as defined before)*

fringe/seen nodes are adjacent to a tree node *(more specific)*

unseen nodes are the rest *(similar as before)*

Starting from one start node x:

1. Add adjacent nodes of x to fringe nodes, setting parent and storing its distance.

Then in each iteration:

1. Add the fringe node f with minimum $distance[f]$ to the tree, and remove it from the fringe.
2. For each adjacent nodes, if it is seen, test if this path is better, and update parent and stored distance;
3. if it is unseen, add to fringe, updating parent and distance

In step 1, starting from the node with minimum distance doesn't guarantee an overall shortest path. A shorter path can be founder later on step 2.
This is like DFS searching. Next adj will be added to the tree only when we finish all the reachable nodes of this node.

7.5.2 Classic Implementation

```
Input: Weighted graph (G,W) together with a pair of nodes start, finish
Output: Length of shortest path from start to finish
tree[start] = true
for x in adj[start]:
  # add x to fringe
  fringe[x] = true
  parent[x] = start
  distance[x] = W[start, x]
while not tree[finish] and fringe nonempty:
  Select a fringe node f with minimum distance[f]
  fringe[f] = false
  tree[f] = true
  for y in adj[f]:
    if not tree[y]:
      if fringe[y]:
        # update distance and candidate arc
        newDistance = distance[f] + W[f, y]
        if newDistance < distance[y]:
          # we found a better path
          distance[y] = newDistance
          parent[y] = f
      else:
        # y is unseen
        fringe[y] = true
        distance[y] = distance[f] + W[f, y]
        parent[y] = f
return distance[finish]
```

We can retrieve the shortest path from the *parent* array: start, $\ldots$, $parent[x], x$.

7.5.3 Implementation with Priority Queue

```
Q = PQcreate()
for x in NodesG:
  key[x] = ∞
```

```
parent[x] = nil
  insert(Q, x)
updateKey(Q,start,0)
while not tree[finish] and not isEmpty(Q):
  f = getMin(Q); deleteMin(Q)
  tree[f] = true
  for y in adj[f]:
    if not tree[y]: # so y in Q
      if key[f]+W[ f, y] < key[y]:
        updateKey(Q, y, key[f] +W[ f, y])
        parent[y] = f
```

7.6 A* Algorithm

7.6.1 Idea

A* algorithm is based on Dijkstra's Algorithm.

It use a **heuristic function h(x)** that underestimates the distance from any node x to the finish node, to replace the trivial distance calculation in Dijkstra's. It chooses a fringe node with minimum heuristic function value rather than the distance.

If we are dealing with cities on a map, h could be the Euclidean distance.

Refer back to Dijkstra's Algorithm for detailed procedures.

A heuristic function is **consistent** if:

- For any adjacent nodes x, y we have $h(x) \leq W(x, y) + h(x)$, and
- $h(\text{finish}) = 0$

A heuristic function is **admissible** if for any node x we have $h(x) \leq$ the weight of the shortest path from x to the goal finish.

7.6.2 Implementation

The blue lines shows difference between Dijkstra's Algorithm.

```
Input: Weighted graph (G,W) together with a pair of nodes start, finish, and
consistent heuristic function h
Output: Length of shortest path from start to finish
tree[start] = true
g[start] = 0
```

```
f[start] = g[start] + h[start]
for x in adj[start]:
  # add x to fringe
  fringe[x] = true
  parent[x] = start
  g[x] = W[start, x] # was `distance[x] = W[start, x]` in Prim's
  f[x] = g[x] + h[x]
while not tree[finish] and fringe nonempty:
  Select a fringe node f such that f[f] is minimum
  fringe[f] = false
  tree[f] = true
  for y in adj[f]:
    if not tree[y]:
      if fringe[y]:
        # update g(y), f(y) and candidate arc
        if g[f] + W[f, y] < g[y]:
          g[y] = g[f] + W[f, y]
          f[y] = g[y] + h[y]
          parent[y] = f
      else:
        # y is unseen
        fringe[y] = true
        g[y] = g[f] + W[f, y]
        f[y] = g[y] + h[y]
        parent[y] = f
return g[finish]
```

We can retrieve the shortest path from the *parent* array: start, $\ldots$, $parent[x], x$, with length $g[x]$.

7.6.3 A* algorithm with priority queues

The blue lines shows difference between Dijkstra's Algorithm.

```
Q = PQcreate()
for x in nodes(G):
  g[x] = ∞
  key[x] = ∞
  parent[x] = null
```

```
  insert(Q, x)
g[start] = 0
updateKey(Q, start, g[start] + h[start])
while not tree[finish] and not isEmpty(Q):
  x = getMin(Q); deleteMin(Q)
  tree[x] = true
  for y in adj[x]:
    if not tree[y]: # so y in Q
      if g[x] + W[x, y] < g[y]:
        g[y] = g[x] + W[x, y]
        updateKey(Q, y, g[y] + h[y])
        parent[y] = x
```

7.7 Warshall's Algorithm

Warshall's Algorithm determines if there is a path from i to j, and combined with Floyd's algorithm to determine the shortest path.

7.7.1 Idea

A path exists between two nodes i and j iff:

- There is an arc from i to j; or
- There is path from i to j going through node v_1; or
- There is path from i to j going through node v_1 and/or v_2; or
- …
- There is a path from i to j going through any of the other nodes.

Now we take an example.
Suppose that the nodes are $\{1, \ldots, n\}$. Consider a path $p = x_1, x_2, \ldots, x_k$ from x_1 to x_k. We say that nodes $x_2, \ldots, x_{k-1}$ are **intermediate nodes** of p.
Let $B_k[i, j] = 1(=$ true) iff there is a path from i to j which uses intermediate nodes $\leq k$ (set $B_k[i, j] = 0(=$ false) otherwise).
Suppose we have a shortest path p from i to j using intermediate nodes $\leq k$ of length d. On each iteration:

- When k is not an intermediate node of p, then $B_{k-1}[i, j]$ already

- When k is an intermediate node of p, then $B_{k-1}[i,k] and B_{k-1}[k,j]$ so $B_{k-1}[i,j]$, since k occurs only once, or we can shorten the path by removing cycles.

7.7.2 Implementation

```
input A
copy A into B (array of Booleans) # B = B_0
for k = 1 to n:
  # B = B_{k-1}
  for i = 1 to n:
    for j = 1 to n:
      b_ij = b_ij or (b_ik and b_kj)
  # B = B_k
# B = B_n
return B
```

7.8 Floyd's Algorithm

This can be solved efficiently using a simple modification of Warshall's algorithm —**Floyd's algorithm**. Let $B_k[i,j]$ = shortest length from i to j, iff there is a path from i to j which uses intermediate nodes $\leq k$ (set $B_k[i,j] = \infty$ otherwise). (Note that previously in Warshall's algorithm, we set it to 0 or 1)
On each iteration, we now have(Blue shows difference with Warshall's):

- When k is not an intermediate node of p, then $d = B_{k-1}[i,j]$ already
- When k is an intermediate node of p, then $d = B_{k-1}[i,k] + B_{k-1}[k,j]$, since k occurs only once, or we can shorten the path by removing cycles.

We see that $d = B_k[i,j] = \min(B_{k-1}[i,j], B_{k-1}[i,k] + B_{k-1}[k,j])$.

7.8.1 Implementation

$$\text{set}B[i,j] = \begin{cases} 0 & \text{if } i = j \\ A[i,j] & \text{if } i \neq j \text{and there is an arc}(i,j) \\ \infty & \text{otherwise} \end{cases}$$

```
input A
copy A into B (array of Booleans) # B = B_0
for k = 1 to n:
```

```
  # B = B_{k-1}
  for i = 1 to n:
    for j = 1 to n:
      b_ij = min(b_ij, b_ik + b_kj)
  # B = B_k
# B = B_n
return B
```

Complexity: $O(n^3)$

7.9 The Travelling Salesman Problem

THE TRAVELLING SALESMAN PROBLEM (TSP) Given a complete weighted graph (G, W), find a way to tour the graph visiting each node exactly once and travelling the shortest possible distance.

TSP involves potentially checking $n!$ different tours if G has n nodes.

7.10 Bellman-Held-Karp Algorithm

Bellman-Held-Karp Algorithm solves the TSP in $O(n^2 2^n)$.

7.10.1 Idea

Let (G, W) have $Nodes = \{1, \ldots, n\}$.
For each $x = 1$ and each $S \subseteq Nodes \setminus \{1, x\}$:

1. Find the min cost $C(S, x)$ of a path from node 1 to node x using exactly S.
2. A tour can be S, that starts from 1 and ends at x, and return from x to 1, see Figure **??**.
3. So solution is
$$\min_{x \neq 1} C(Nodes \setminus \{1(\text{node } 1), x\}, x) + W(x, 1)$$
.

Suppose node y is the first node in S that lies before x, as in Figure **??**, then

$$C(S, x) = \min_{y \in S} C(S \setminus y, y) + W(y, x)$$

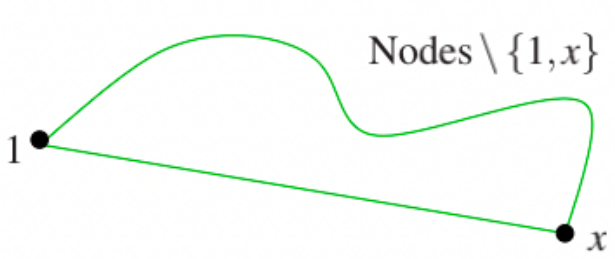

Figure 7: Bellman-Help-Karp Algorithm

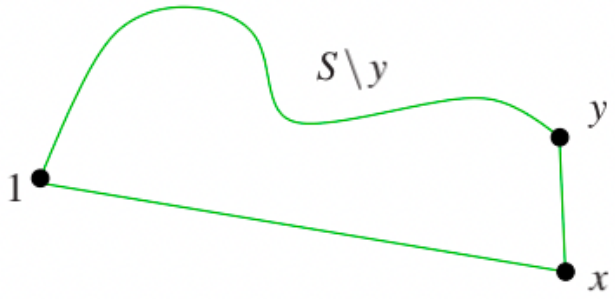

Figure 8: Bellman-Help-Karp Algorithm

7.10.2 Implementation

```
Input (G,W)
Choose start ∈ NodesG
for x ∈ Nodes \ {start}:
  C[∅, x] = W[start, x] # Process sets S in increasing order of size.
for S ⊆ Nodes \ {start} with S = ∅:
  for x ∈ Nodes \ (S ∪ {start}):
    # Find C[S, x]
    C[S, x] = ∞
    for y ∈ S:
       C[S, x] = min(C[S \ {y}, y] + W[y, x], C[S, x])
# Now have calculated and stored all values of C[S, x]
opt = ∞
for x ∈ Nodes \ {start}:
  opt = min(C[Nodes \ {start, x}, x] + W[x, start], opt)
return opt
```

Remark: The Bellman-Held-Karp algorithm can be adapted to solve the Hamiltonian circuit problem. The complexity is still $O(n^2 2^n)$.

8 Introduction

8.1 Searching an unordered list

8.1.1 Linear Search

```
k = 0
while k < n:
  if L[k] == x:
    return k
  else:
    k = k + 1
return "not found"
```

LS is trivially minimal, since in the worst case, we must have inspected every item exactly once.

8.2 Searching an ordered list

8.2.1 Modified Linear Search

Since our list is sorted now, we can stop searching if we found some item already greater than the target.

```
k = 0
while k < n:
  cond:
    L[k] = x: return k
    L[k] > x: return "not found"
    L[k] < x: k = k +1
return "not found"
```

Worst case complexity is still $O(n)$

8.2.2 Binary Search

Binary Search reduces the data into a half in each iteration.

```
procedure BinSearch(left,right):
  # searches for x in L in the range L[left] to L[right]
  if left > right:
    return "not found"
```

```
else:
  mid :=  (left+right)/2
cond
  x = L[mid]: return mid
  x < L[mid]: return BinSearch(left, mid - 1)
  x > L[mid]: return BinSearch(mid + 1, right)
```

8.3 Analysing complexity

Worse-case analysis Let $W(n)$ be the largest number of comparisons made when performing the algorithm on the whole range of inputs of size n.

Worse-case analysis Let $A(n)$ be the average number of .

8.3.1 Analysing complexity of Binary Search

First, write down the recurrence relation:

$$W(1) = 1 W(n) = 1 + W(\lfloor n/2 \rfloor)$$

Solving $W(n)$ by repeated expansion:

$$\begin{aligned} W(n) &= 1 + W(\lfloor n/2 \rfloor) \\ &= 1 + 1 + W(\lfloor n/4 \rfloor) \\ &\dots \\ &= 1 + 1 + \ldots + W(1) \end{aligned}$$

The number of 1s is the number of times we can divide n by 2. So we have:

$$W(n) = \lfloor \log n \rfloor + 1$$

We can now represent the algorithm by a binary decision tree, with n nodes. See Figure **??**. Let d be the depth of the tree. ($d = \lfloor \log n \rfloor$) Worst case performance will be $d + 1$.

8.3.2 Lower bound for searching

LOWER BOUND FOR SEARCHING Any algorithm for searching an ordered list of length n for element x, and which only accesses the list by comparing x with entries in the list, must perform at least $\lfloor \log n \rfloor + 1$ comparisons in worst case.

So Binary Search is optimal.

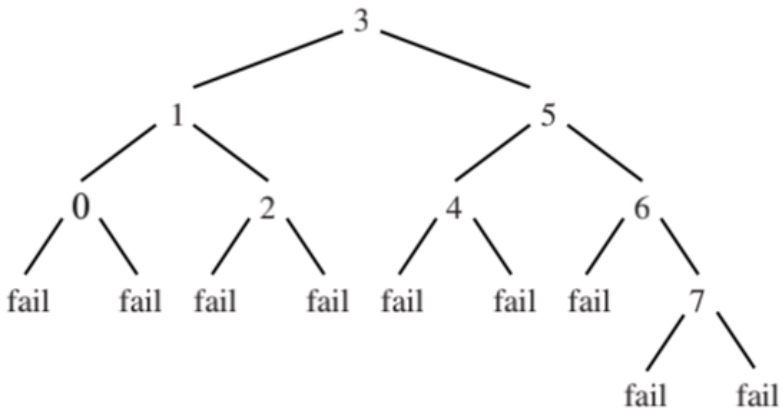

Figure 9: Decision tree for Binary Search ($n = 8$)

9 Orders of Complexity

9.1 Definition of orders

Various possible orders and examples:

polynormial 1 (**constant**), n (**linear**), n^2 (**quadratic**), …

exponential 2^n, 3^n

logarithmic $\log n$

log linear $n \log n$

9.1.1 Order Notations

f is $O(g)$ ("f is big Oh of g") iff there are $m \in \mathbb{N}$, $c \in \mathbb{R}^+$ such that for all $n \geq m$ we have $f(n) \leq c \cdot g(n)$.
(f is *bounded* by g, i.e. order of f is $\leq$ order of g)

f is $\theta(g)$ ("f is order g") iff f is $O(g)$ and g is $O(f)$. (f has the same order as g)

10 Sorting

10.1 Lower Bound for Sorting by Comparison

A node of a tree is a **leaf** if it has no successor nodes. Otherwise, it is **internal**.

Proposition If a binary tree has depth d then it has $\leq 2^d$ leaves.

A binary tree of depth d is **balanced** if every leaf node is of depth d or $d - 1$.

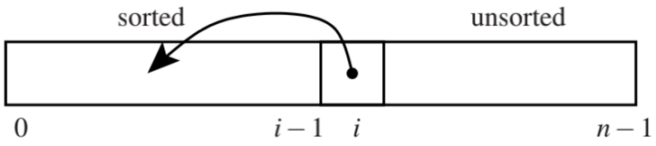

Figure 10: Insertion Sort

Proposition If a binary tree is unbalanced then we can find a balanced tree with the same number of leaves without increasing the total path length.

Any algorithm for sorting a list of length n by comparisons must perform at least $\lfloor \log(n!) \rfloor$ comparisons in average case, and this is almost the same for the worst case.
Proof can be found on Lecture Notes page 69.

10.2 Insertion sort

Suppose that $L[0..1-1]$ is known to be sorted(where $1 \leq i < n$). Then insert $L[i]$ into $L[0..i-1]$ in its correct position. When this is done, $L[0..i]$ will be sorted. See Figure **??**.

10.2.1 Implementation

```
i = 1
while i < n:
  # insert L[i] into L[0..i-1]
  j = i
  while L[j - 1] > L[ j] and j > 0:
    Swap(j - 1, j)
    # swaps L[j - 1] and L[j]. L[ j] will always be the value to be inserted
    j = j - 1
  i = i + 1
```

10.2.2 Complexity of Insertion Sort

The insertion can be done by comparing $L[i]$ successively with $L[i-1]$, $L[i-2]$, ... until we find the first $L[j]$ such that $L[i]$ $L[j]$. So in worse case:

$$W(n) = \sum_{i}^{n-1} i = \frac{n(n-1)}{2}$$

So Insertion Sort is $\theta(n^2)$, it is not optimal.

10.3 Mergesort

```
procedure Mergesort(array, left, right):
  if left < right:
    mid = ⌊(left + right) / 2⌋
    # left ≤ mid < right
    Mergesort(array, left, mid)
    Mergesort(array, mid + 1, right)
    # Now sublists L[left..mid] and L[mir + 1..right] are sorted
    Merge(array, left, mid, right) # Merges the sublists in-place into array
```

The procedure Merge generate the merged list in ascending order by repeatedly removing the current least value from the two lists to be merged. Since the lists to be merged are sorted, it is enough to compare the leftmost elements of each list until one of the lists is exhausted, after which the remainder of the other list is transferred automatically.

10.3.1 Analysing complexity of Mergesort

We see from the procedure Merge that each element is placed in the merged list at the expense of at most one comparison, apart from the last element, which can always be transferred automatically. So no of comparisons is length -1, i.e. right $-$ left.
It can be shown that this form of merge is optimal in worst case.
Now we obtain the recurrence relation:

$$W(1) = 0$$
$$W(n) = n - 1 + W(\lceil n/2 \rceil) + W(\lfloor n/2 \rfloor)$$

Assuming $n = 2^k$ for simplicity, we have:

$$\begin{aligned}
W(n) &= n - 1 + 2W(n/2) \\
&= (n-1) + 2(n/2 - 1) + 2^2 W(n/2^2) \\
&= n + n - (1+2) + 2^2 W(n/2^2) \\
&\dots \\
&= n + n + \ldots + n - (1 + 2 + 2^2 + \ldots + 2^{k-1}) + 2^k W(n/2^k) \\
&= k \cdot n - (n-1) && \text{[Simplify]} \\
&= n \log(n) - n + 1 && \text{[k = log n]} \\
&= n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 && \text{[n is power of 2]}
\end{aligned}$$

Now $W(n)$ for Mergesort has the same order as the lower bound $\log(n!)$. Notice that for large n:

$$\begin{aligned}\log(n!) &= \sum_{i-1}^{n} \log i \\ &\approx \int_1^n \log x \, dx \\ &= [x \ln(x) - x]_1^n \\ &= n \ln(n) - n + 1\end{aligned}$$

10.4 Master Theorem

Ignored. Can be found on Lecture Nodes page 71–75.

10.5 Quick sort

Lecture Nodes page 75.

10.6 Heapsort

heap is a left-complete binary tree. Left-complete means that if the tree has depth d then

- all nodes are present at depth 0, 1, ..., $d-1$;
- at depth d no node is missing to the left of a node which is present.

A tree is **minimising partial order tree** if the key at any node $\leq$ the keys at each child node(if any).

min heap is a heap that is a minimising partial order tree.

A tree is **minimising partial order tree** if the key at any node $\geq$ the keys at each child node(if any).

max heap is a heap that is a maximising partial order tree.

10.6.1 Implementation

See Lecture Nodes on page 80.

Introduction to Complexity

11 Tractable problems and P

tractable/feasible/efficiently computable means can be computed in a reasonable amount of time.

(worse-case)

Example:

- Sorting a list by comparisons is tractable.
- The EulerPath problem (finding Euler path) is tractable.
- The HamPath problem is intractable since it is $O(n!)$.

11.1 Decision problems

decision problem A **decision problem** D is decided by an algorithm A if for any input x, A returns 'yes' or 'no' depending on whether $D(x)$ (and in particular A always terminates).

decidable A decision problem is **decidable** in polynomial time (poly time or p-time) iff it is decided by some algorithm A which runs within polynomial time.

The EulerPath is decidable in p-time, but HamPath is not.
Informally, 'tractable' means 'polynomial time'.

Cook-Karp Thesis A problem is tractable if it can be computed within polynomially many steps in worst case.

Polynomial invariance thesis If a problem can be solved in p-time in some reasonable model of computation, then it can be solved in p-time in any other reasonable model of computation.

complexity class P A decision problem $D(x)$ is in the complexity class P (p-time) if it can be decided with time $p(n)$ in some reasonable model of computation.

Proposition Suppose that f and g are functions which are p-time computable. Then the composition $g \circ f$ is also p-time computable.

This proposition can be useful for reduction and NP-completeness.

12 The complexity class NP

NP A decision problem $D(x)$ is in **NP** (non-deterministic polynomial time) if there is a problem $E(x, y)$ in P and a polynomial $p(n)$ such that

- $D(x)$ iff $\exists y.E(x, y)$
- if $E(x, y)$ then $|y| \leq p(|x|)$ (E is polynomially balanced)

Where y must be polynomially bounded otherwise it would take too long to guess y.

12.1 Determining HamPath in NP

Break HamPath problem into a guessing and checking solutions. So

$$\text{HamPath}(G)\text{iff}\exists\pi.\text{Ver-HamPath}(G,\pi)$$

Where Ver-HamPath is the sub problem that verifies whether the solution is correct.
Clearly Ver-HanPath(G,π) can be p-bounded in the size of G. So HamPath $\in$ NP.

12.1.1 Remark

- P class of decision problems which can be efficiently *solved*
- NP class of decision problems which can be efficiently *verified*

If a decision problem is in P then it is in NP, i.e. P NP.

13 Problem reduction

Suppose that the easier decision problem D and the harder decision problem D'.

reduction D **(many-one) reduces** to D' if there is a p-time computable function f such that $D(x)$ iff $D'(f(x))$. Note that f can be a many-one function.

We can reduce a question about D to a question about D'. Then

Proposition 1 Suppose D $\leq$ D' and D' $\in$ P. Then D $\in$ P.

Proposition 2 Suppose D $\leq$ D' and D' $\in$ NP. Then D $\in$ NP.

The reduction order is reflexive and transitive. If both D $\leq$ D' and D' $\leq$ D we write D D'. Here D and D' are as hard as each other.

14 NP-completeness

NP-hard A decision problem D is **NP-hard** if for all problems $D' \in NP$ we have $D' \leq D$.

Thus NP-hard problems are at least as hard as all NP problems. Note that NP-hard problems do not necessarily belong to NP. They could be harder.
If D is NP-hard and $D \leq D'$ then D' is also NP-hard (by transitivity of reduction).

NP-complete A decision problem D is **NP-complete** (NPC) if

- $D \in$ NP
- D is NP-hard

NP-complete problems are the hardest problems in NP.

14.1 Intractability via NP-completeness

Suppose P $\neq$ NP. If a problem D is NP-hard then D $\notin$ P.

14.2 Proving NPC

To see that problem D is NPC show:

1. $D \in$ NP (typically using guess and verify).
2. $D' \leq D$ for some known NPC problem D'. (To establish D is NP-hard)

As an example take HamPath. We have already seen that HamPath $\in$ NP by guessing and verifying in p-time. If we can show SAT $\leq$ HamPath then we can conclude that HamPath is NPC. It is indeed possible to show SAT $\leq$ HamPath but we omit the reduction as it is long and difficult. So HamPath is NP-complete.