CSE 421: Introduction to Algorithms

Greedy Algorithms

Winter 2005
Paul Beame

Greedy Algorithms

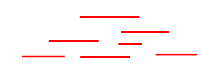
- Hard to define exactly but can give general properties
  - Solution is built in small steps
  - Decisions on how to build the solution are made to maximize some criterion without looking to the future
    - Want the 'best' current partial solution as if the current step were the last step
- May be more than one greedy algorithm using different criteria to solve a given problem

Greedy Algorithms

- Greedy algorithms
  - Easy to produce
  - Fast running times
  - Work only on certain classes of problems
- Two methods for proving that greedy algorithms do work
  - Greedy algorithm stays ahead
    - At each step any other algorithm will have a worse value for the criterion
  - Exchange Argument
    - Can transform any other solution to the greedy solution at no loss in quality

Interval Scheduling

- Interval Scheduling
  - Single resource
  - Reservation requests



    - Of form "Can I reserve it from start time **s** to finish time **f**?"
    - **s < f**
  - **Find:** maximum number of requests that can be scheduled so that no two reservations have the resource at the same time

Interval scheduling

- Formally
  - Requests **1**,**2**,…,**n**
    - request **i** has start time s_i and finish time $f_i > s_i$
  - Requests **i** and **j** are **compatible** iff either
    - request **i** is for a time entirely before request **j**
      - $f_i \le s_j$
    - or, request **j** is for a time entirely before request **i**
      - $f_j \le s_i$
  - Set **A** of requests is **compatible** iff every pair of requests $i,j \in A$, $i \neq j$ is compatible
  - Goal: Find maximum size subset **A** of compatible requests

Greedy Algorithms for Interval Scheduling

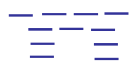
- What criterion should we try?
  - Earliest start time s_i
  - Shortest request time $f_i - s_i$
  - Earliest finish fime f_i

Greedy Algorithms for Interval Scheduling

- What criterion should we try?
  - Earliest start time s_i
    - Doesn't work

  - Shortest request time $f_i - s_i$
    - Doesn't work

  - Even fewest conflicts doesn't work



  - Earliest finish time f_i
    - Works

Greedy Algorithm for Interval Scheduling

R←set of all requests
A←∅
While R≠∅ do
  Choose request **i**∈**R** with smallest finishing time f_i
  Add request **i** to **A**
  Delete all requests in **R** that are not compatible with request **i**
Return **A**

Greedy Algorithm for Interval Scheduling

- Claim: **A** is a compatible set of requests and these are added to **A** in order of finish time
  - When we add a request to **A** we delete all incompatible ones from **R**
- Claim: For any other set **O**⊆**R** of compatible requests then if we order requests in **A** and **O** by finish time then for each **k**:
  - If **O** contains a k^{th} request then so does **A** and
  - the finish time of the k^{th} request in **A**, is ≤ the finish time of the k^{th} request in **O**, i.e. "$a_k \le o_k$" where a_k and o_k are the respective finish times

Inductive Proof of Claim: $a_k \le o_k$

- Base Case: This is true for the first request in **A** since that is the one with the smallest finish time
- Inductive Step: Suppose $a_k \le o_k$
  - By definition of compatibility
    - If **O** contains a **k+1**st request **r** then the start time of that request must be after o_k and thus after a_k
    - Thus **r** is compatible with the first **k** requests in **A**
    - Therefore
      - **A** has at least **k+1** requests since a compatible one is available after the first **k** are chosen
      - **r** was among those considered by the greedy algorithm for that **k+1**st request in **A**
    - Therefore by the greedy choice the finish time of **r** which is o_{k+1} is at least the finish time of that **k+1**st request in **A** which is a_{k+1}

Implementing the Greedy Algorithm

- Sort the requests by finish time
  - **O**(**nlog n**) time
- Maintain current latest finish time scheduled
- Keep array of start times indexed by request number
- Only eliminate incompatible requests as needed
  - Walk along array of requests sorted by finish times skipping those whose start time is before current latest finish time scheduled
  - **O**(**n**) additional time for greedy algorithm

Exchange arguments

- Scheduling to minimize lateness
  - Single resource as in interval scheduling but instead of start and finish times request **i** has
    - Time requirement t_i which must be scheduled in a contiguous block
    - Target deadline d_i by which time the request would like to be finished
    - Overall start time **s**
  - Requests are scheduled by the algorithm into time intervals $[s_i, f_i]$ such that $t_i = f_i - s_i$
  - Lateness of schedule for request **i** is
    - If $d_i < f_i$ then request **i** is late by $L_i = f_i - d_i$ otherwise its lateness $L_i = 0$
  - Maximum lateness $L = \max_i L_i$
  - **Goal:** Find a schedule for **all** requests (values of s_i and f_i for each request **i**) to minimize the maximum lateness, **L**

Greedy Algorithm: Earliest Deadline First

- Order requests in increasing order of deadlines
- Schedule the request with the earliest deadline as soon as the resource becomes available

13

Greedy Algorithm: Earliest Deadline First

Sort deadlines in increasing order
  (assume wlog that $\mathbf{d_1 \le d_2 \le \ldots \le d_n}$)
$\mathbf{f} \leftarrow \mathbf{s}$
for $\mathbf{i} \leftarrow \mathbf{1}$ to $\mathbf{n}$ to
    $\mathbf{s_i} \leftarrow \mathbf{f}$
    $\mathbf{f_i} \leftarrow \mathbf{s_i + t_i}$
    $\mathbf{f} \leftarrow \mathbf{f_i}$
end for

14

Why does this give optimal value of maximum lateness?

- Easy observations
  - This schedule has no idle time
    - The overall schedule starts at time **s** and finishes as soon as possible at time **$s+t_1+t_2+\ldots+t_n$**
  - There is an optimal schedule with no idle time
    - Shifting the requests earlier by the amount of the idle time can only decrease maximum lateness
    (It might not improve the maximum lateness but it certainly can't hurt to do this.)
    - There are only a finite # of schedules with no idle time so one of them must be best

15

Exchange Argument

- We will show that if there is another schedule **O** (think optimal schedule) then we can gradually change **O** so that
  - at each step the maximum lateness in **O** never gets worse
  - it eventually becomes the same as **A**

16

Inversions

- Inversion
  - $\mathbf{d_j < d_i}$ but **i** is scheduled before **j**
  - Earliest deadline first has no inversions
- Claim: All schedules with no inversions and no idle time have the same maximum lateness
- Proof
  - Schedules can differ only in how they order requests with equal deadlines
  - Consider all requests having some common deadline **d**
  - Maximum lateness of these jobs is based only on the finish time of the last of these jobs but the set of these requests occupy the same time segment in both schedules
    - Last of these requests finishes at the same time in any such schedule.

17

Optimal schedules and inversions

- Claim: There is an optimal schedule with no idle time and no inversions
- Proof:
  - By previous argument there is an optimal schedule **O** with no idle time
  - (i) If **O** has an inversion then it has a **consecutive** pair of requests in its schedule that are inverted
    i.e. $\mathbf{d_j < d_i}$ but **i** is scheduled immediately before **j**
    - (Simple transitivity of <, otherwise consecutive pairs would always be scheduled in increasing order of deadlines)

18

Optimal schedules and inversions

- (ii) If $d_j < d_i$ but **i** is scheduled in **O** immediately before **j** then swapping requests **i** and **j** to get schedule **O'** does not increase the maximum lateness
  - Lateness $L_j' \leq L_j$ since **j** is scheduled earlier in **O'** than in **O**
  - Requests **i** and **j** together occupy the same total time slot in both schedules
    - All other requests $k \neq i$ have $L_k' = L_k$
    - $f_i' = f_j$ so $L_i' = f_j - d_i < f_j - d_j = L_j$
  - Maximum lateness has not increased!

19

Optimal schedules and inversions

- (iii) Eventually these swaps will produce an optimal schedule with no inversions
  - Each swap decreases the number of inversions by **1**
  - There are at most **n**(**n-1**)/**2** inversions

QED

20

Earliest Deadline First is optimal

- We know that
  - There is an optimal schedule with no idle time or inversions
  - All schedules with no idle time or inversions have the same maximum lateness
  - EDF produces a schedule with no idle time or inversions
- Therefore
  - EDF produces an optimal schedule

21

Optimal Caching/Paging

- Memory systems
  - many levels of storage with different access times
  - smaller storage has shorter access time
  - to access an item it must be brought to the lowest level of the memory system
- Consider the management problem between adjacent levels
  - Main memory with **n** data items from a set **U**
  - Cache can hold **k<n** items
  - Simplest version with no direct-mapping or other restrictions about where items can be
  - Suppose cache is full initially
    - Holds **k** data items to start with

22

Optimal Caching/Paging

- Given a memory request **d** from **U**
  - If **d** is stored in the cache we can access it quickly
  - If not then we call it a cache miss and (since the cache is full)
    - we must bring it into cache and evict some other data item from the cache
    - which one to evict?
- Given a sequence $D = d_1, d_2, \ldots, d_m$ of elements from **U** corresponding to memory requests
- Find a sequence of evictions (an eviction schedule) that has as few cache misses as possible

23

Caching Example

- **n**=**3**, **k**=**2**, **U**={**a**,**b**,**c**}
- Cache initially contains {**a**,**b**}
- **D**= **a b c b c a b**
- **S**= **a** **c**
- **C**= $\boxed{\begin{matrix}a\\b\end{matrix}}$ $\boxed{\begin{matrix}b\\c\end{matrix}}$ $\boxed{\begin{matrix}a\\b\end{matrix}}$
- This is optimal

24

A Note on Optimal Caching

- In real operating conditions one typically needs an on-line algorithm
  - make the eviction decisions as each memory request arrives
- However to design and analyze these algorithms it is also important to understand how the best possible decisions can be made if one did know the future
  - Field of on-line algorithms compares the quality of on-line decisions to that of the optimal schedule
- What does an optimal schedule look like?

Belady's Greedy Algorithm: Farthest-In-Future

- Given sequence $\mathbf{D}=\mathbf{d}_1,\mathbf{d}_2,\ldots,\mathbf{d}_m$
- When $\mathbf{d}_i$ needs to be brought into the cache evict the item that is needed farthest in the future
  - Let **NextAccess**$_i$(**d**)=**min**{ **j≥i : d**$_j$**=d**} be the next point in **D** that item **d** will be requested
  - Evict **d** such that **NextAccess**$_i$(**d**) is largest

Other Algorithms

- Often there is flexibility, e.g.
  - k=3, C={a,b,c}
  - D= a b c d a d e a d b c
  - S_{FIF}= c b e d
  - S = b c d e
- Why aren't other algorithms better?
  - Least-Frequenty-Used-In-Future?
- Exchange Argument
  - We can swap choices to convert other schedules to Farthest-In-Future without losing quality

Reduced Schedule

- We seemed to assume that we only brought in new items (and evicted current items) at cache misses
  - Would acting in advance would help?
- Call an eviction schedule reduced if its only evictions are of single items at cache misses
  - Farthest-In-Future produces a reduced schedule
- Given an eviction schedule **S** define schedule **reduced**(**S**) as follows:
  - In step **i** if **S** brought in item **d** that is not accessed until step **j > i** and evicted item e from cache then 'pretend' to do this eviction leaving **d** in main memory until step **j**
- Claim: **reduced**(**S**) yields at most as many cache misses as **S** does

Optimal Caching

- Fix some sequence $\mathbf{D}=\mathbf{d}_1,\mathbf{d}_2,\ldots,\mathbf{d}_m$ of requests
- Let $\mathbf{S}_{\mathbf{FIF}}$ be the schedule produced on this sequence by Farthest-In-Future
- **Claim:** If **S** is any reduced schedule (for all of **D**) that agrees with $\mathbf{S}_{\mathbf{FIF}}$ for (at least) the first **j** steps then
  - there is a reduced schedule **S'** that agrees with $\mathbf{S}_{\mathbf{FIF}}$ for (at least) the first **j+1** steps such that **#misses**(**S'**) ≤ **#misses**(**S**)
- Applying this claim **m** times we could start with any optimal reduced schedule $\mathbf{S}_0$ (which agrees with $\mathbf{S}_{\mathbf{FIF}}$ for at least **0** steps) and produce $\mathbf{S}_{\mathbf{FIF}}$ without increasing the number of cache misses

Proving the Claim

- Suppose **S** is a reduced schedule that agrees with $\mathbf{S}_{\mathbf{FIF}}$ for the first **j** steps
- At the time of the **j+1**st request $\mathbf{d}_{j+1}$, **S** and $\mathbf{S}_{\mathbf{FIF}}$ yield the same cache contents
- Case 1: $\mathbf{d}_{j+1}$ is in the cache
  - Since both **S** and $\mathbf{S}_{\mathbf{FIF}}$ are reduced neither has a miss so they agree for j+1 steps and we can set **S'=S**
- Case 2: $\mathbf{d}_{j+1}$ is not in the cache
  - Say that **S** evicts **f** and $\mathbf{S}_{\mathbf{FIF}}$ evicts **e**
    - If **e=f** then we can take **S'=S** since **S** and $\mathbf{S}_{\mathbf{FIF}}$ agree for **j+1** steps
    - What if **e≠f?**

Building S’

- **NextAccess$_{j+1}$(e)**>**NextAccess$_{j+1}$(f)**
- Define schedule **S’** that agrees with **S** except that
  - At step **j+1** it evicts **e** instead of **f**
  - The first time after step **j+1** that one of the following occurs it does something different that will result in **S’** having the same cache as S so that the rest of the simulation can take place
    - **Event A:** There is a request to an item **g** other than **e** or **f** s.t. **S** evicts **e**
    - **Event B:** There is a request to **f**
  - **Note: S’** will have trouble handling any access to **e** while it remains in the cache under S but we won’t have to deal with it because **Event B** will happen before any access to **e**

Building S’

- **Event A:** There is a request to an item **g** other than **e** or **f** s.t. **S** evicts **e**
  - In this case the cache under **S’** also doesn’t contain **g** (since the caches under **S** and **S’** can at most disagree on **e** and **f**)
    - So we have **S’** evict **f**
    - Caches now agree and **S’** is exactly the same as **S** for the rest
    - **#misses(S’)=#misses(S).**

Building S’

- **Event B:** There is a request to **f**
  - Now **f** is in the cache under **S’** but not **S**
  - Say **S** evicts item **e’**
  - If **e’=e** then after this step the caches under **S’** and **S** are the same so we let **S’** match **S** for the rest
  - **#misses(S’)<#misses(S)**
  - If **e’≠e** then have **S’** evict **e’** also and bring in **e** instead
    - Now cache under **S’** and **S** agree and S’ can match **S**
    - **#misses(S’)=#misses(S)**
    - But **S’** is not reduced so we replace **S’** by **reduce(S’)**
    - **Reduce(S’)** still agrees with S_{FIF} for **j+1** steps

Single-source shortest paths

- Given an (un)directed graph G=(V,E) with each edge e having a non-negative weight w(e) and a vertex v
- Find length of shortest paths from v to each vertex in G

A greedy algorithm

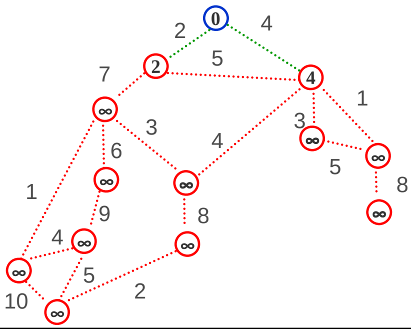
- Dijkstra’s Algorithm:
  - Maintain a set **S** of vertices whose shortest paths are known
    - initially **S**={**s**}
  - Maintaining current best lengths of paths that only go through **S** to each of the vertices in **G**
    - path-lengths to elements of **S** will be right, to **V-S** they might not be right
  - Repeatedly add vertex **v** to **S** that has the shortest path-length of any vertex in **V-S**
    - update path lengths based on new paths through v

Dijsktra’s Algorithm

Dijkstra(**G**,**w**,**s**)
  S←{**s**}
  d[**s**]←**0**
  while **S≠V** do
    of all edges **e**=(**u**,**v**) s.t. **v∉S** and **u∈S** select* one with the minimum value of **d**[**u**]+**w**(**e**)
      S←**S**∪ {**v**}
      d[**v**]←**d**[**u**]+**w**(**e**)
      pred[**v**]←**u**

*For each v∉S maintain d’[v]=minimum value of d[u]+w(e) over all vertices u∈S s.t. e=(u,v) is in of G

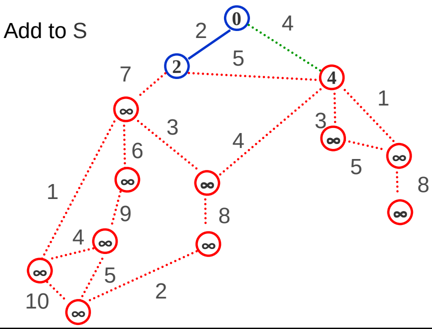
Dijkstra's Algorithm



37

Dijkstra's Algorithm

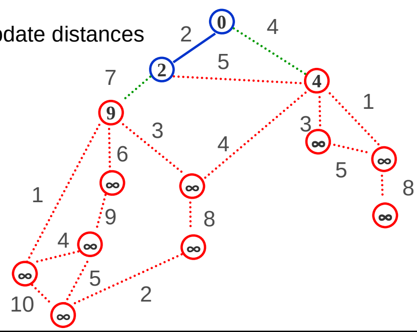
Add to S



38

Dijkstra's Algorithm

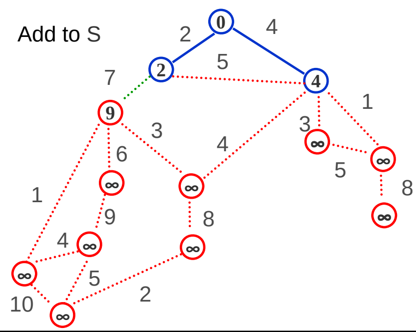
Update distances



39

Dijkstra's Algorithm

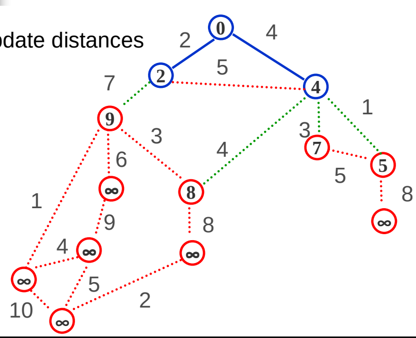
Add to S



40

Dijkstra's Algorithm

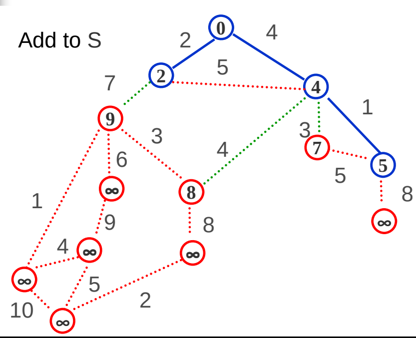
Update distances



41

Dijkstra's Algorithm

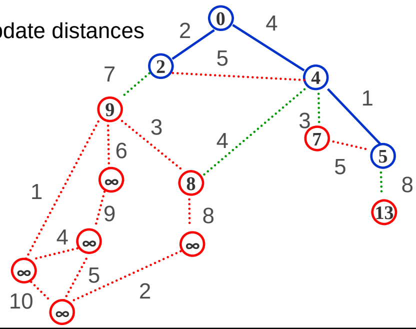
Add to S



42

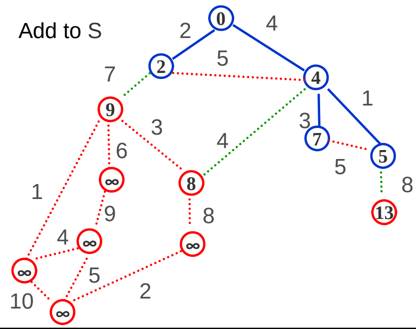
Dijkstra's Algorithm

Update distances



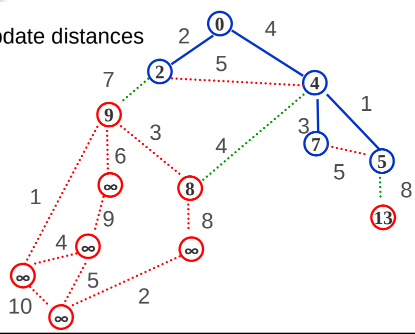
Dijkstra's Algorithm

Add to S



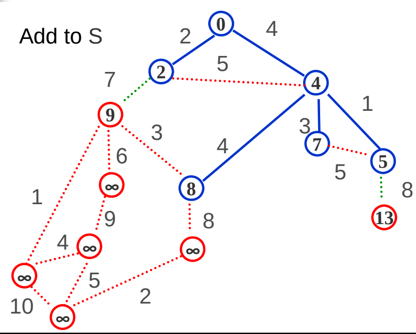
Dijkstra's Algorithm

Update distances



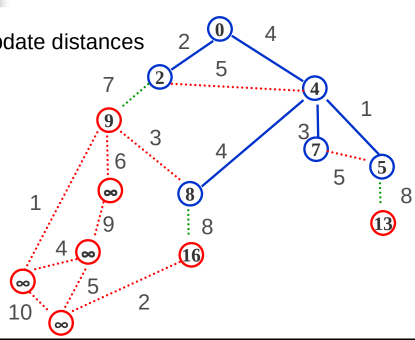
Dijkstra's Algorithm

Add to S



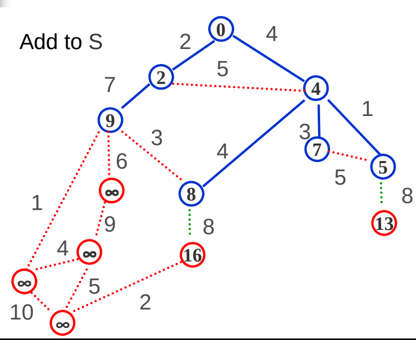
Dijkstra's Algorithm

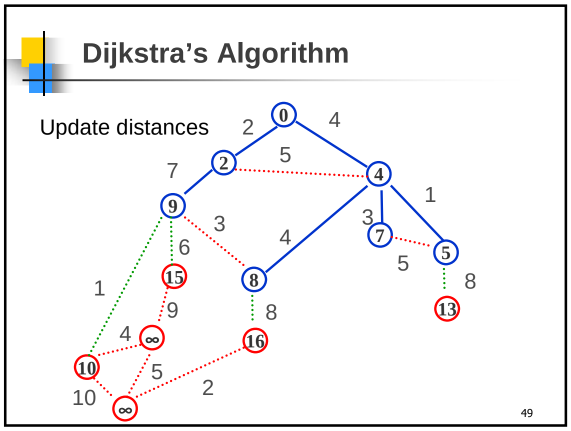
Update distances

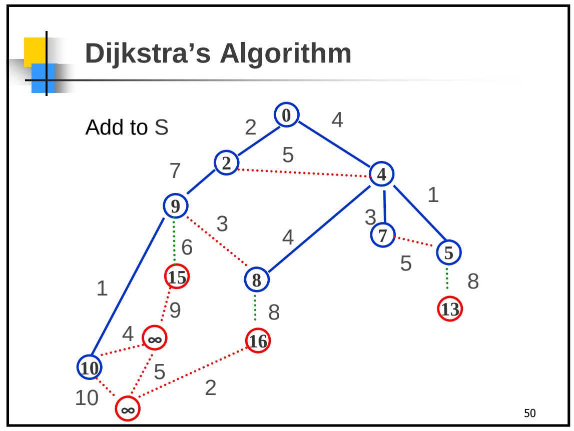


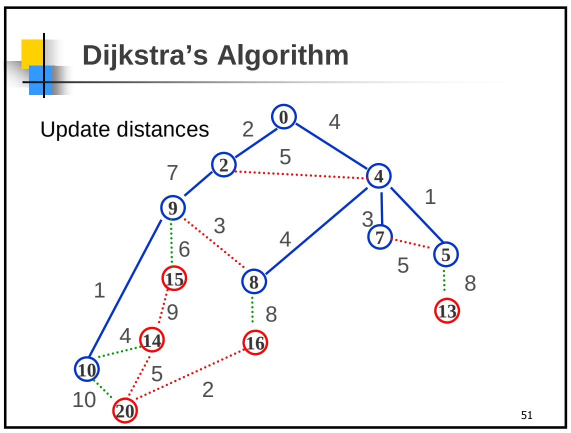
Dijkstra's Algorithm

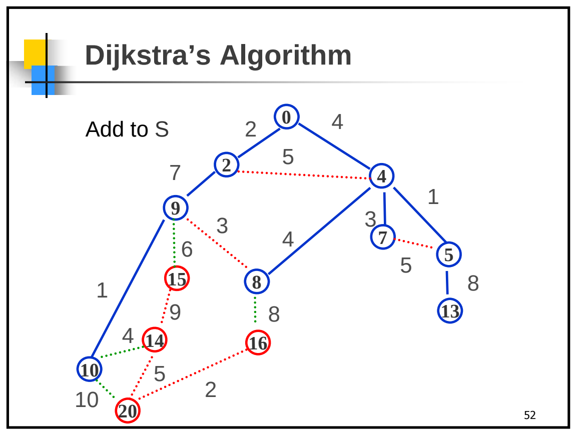
Add to S

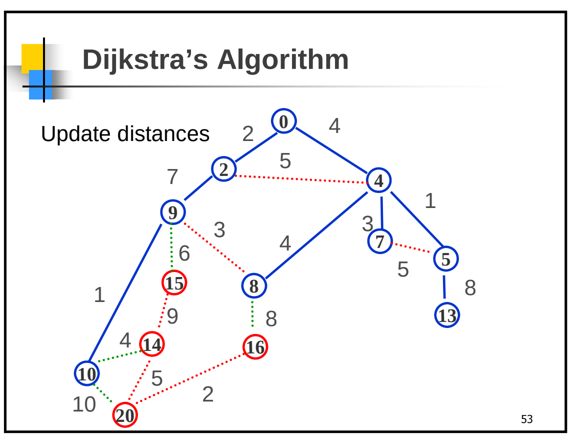


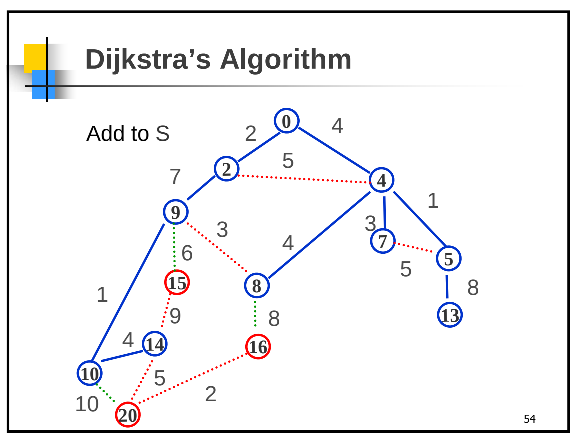

Dijkstra’s Algorithm
Update distances
0
2
4
9
7
5
15
8
13
∞
16
10
∞
49


Dijkstra’s Algorithm
Add to S
0
2
4
9
7
5
15
8
13
∞
16
10
∞
50


Dijkstra’s Algorithm
Update distances
0
2
4
9
7
5
15
8
13
14
16
10
20
51

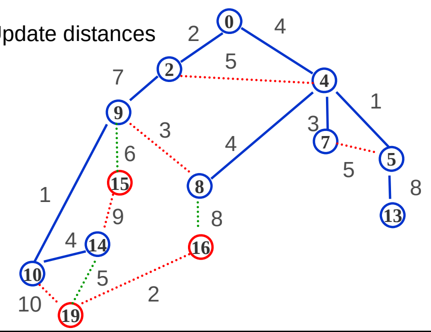

Dijkstra’s Algorithm
Add to S
0
2
4
9
7
5
15
8
13
14
16
10
20
52


Dijkstra’s Algorithm
Update distances
0
2
4
9
7
5
15
8
13
14
16
10
20
53


Dijkstra’s Algorithm
Add to S
0
2
4
9
7
5
15
8
13
14
16
10
20
54

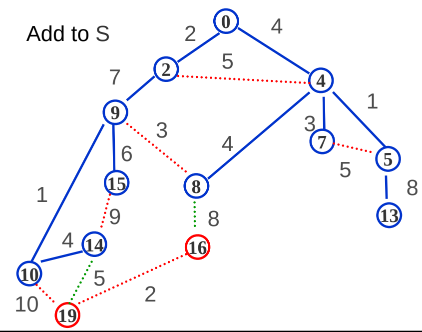
Dijkstra's Algorithm

Update distances



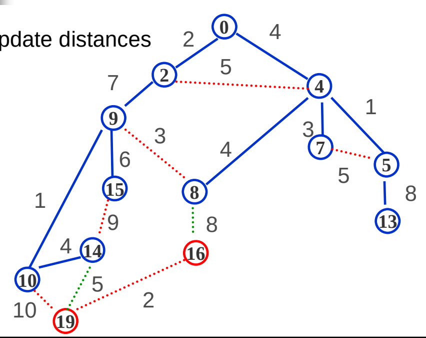
Dijkstra's Algorithm

Add to S



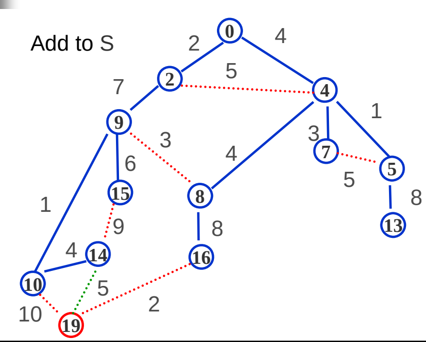
Dijkstra's Algorithm

Update distances



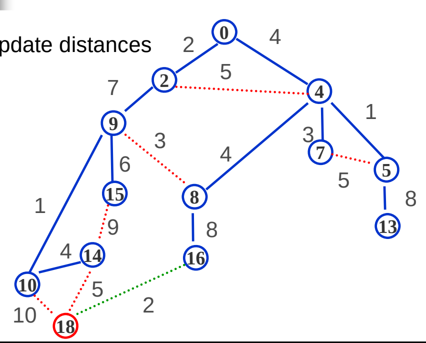
Dijkstra's Algorithm

Add to S



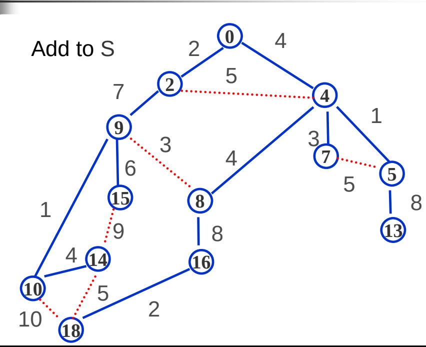
Dijkstra's Algorithm

Update distances



Dijkstra's Algorithm

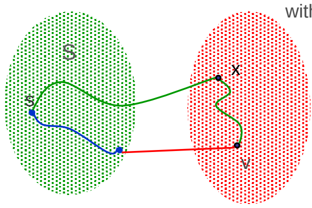
Add to S



Dijkstra's Algorithm Correctness

Suppose all distances to vertices in S are correct and u has smallest current value in V-S

∴distance value of vertex in V-S=length of shortest path from s with only last edge leaving S



Suppose some other path to v and x= first vertex on this path not in S

d'(v)≤ d'(x)

x-v path length ≥ 0

∴ other path is longer

Therefore adding v to S keeps correct distances

Dijkstra's Algorithm

- Algorithm also produces a tree of shortest paths to v following pred links
  - From **w** follow its ancestors in the tree back to **v**
- If all you care about is the shortest path from v to w simply stop the algorithm when w is added to S

Implementing Dijkstra's Algorithm

- Need to
  - keep current distance values for nodes in **V-S**
  - find minimum current distance value
  - reduce distances when vertex moved to **S**

Data Structure Review

- **Priority Queue:**
  - Elements each with an associated **key**
  - Operations
    - **Insert**
    - **Find-min**
      - Return the element with the smallest key
    - **Delete-min**
      - Return the element with the smallest key and delete it from the data structure
    - **Decrease-key**
      - Decrease the key value of some element
- Implementations
  - Arrays: **O**(**n**) time find/delete-min, **O**(**1**) time insert/decrease-key
  - Heaps: **O**(log **n**) time insert/decrease-key/delete-min, **O**(**1**) time find-min

Dijkstra's Algorithm with Priority Queues

- For each vertex **u** not in tree maintain cost of current cheapest path through tree to **u**
  - Store **u** in priority queue with key = length of this path
- Operations:
  - n-1 insertions (each vertex added once)
  - n-1 delete-mins (each vertex deleted once)
    - pick the vertex of smallest key, remove it from the priority queue and add its edge to the graph
  - <m decrease-keys (each edge updates one vertex)

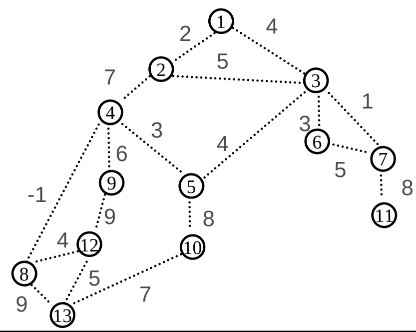
Dijsktra's Algorithm with Priority Queues

- Priority queue implementations
  - Array
    - insert O(1), delete-min O(n), decrease-key O(1)
    - total $O(n+n^2+m)=O(n^2)$
  - Heap
    - insert, delete-min, decrease-key all O(log n)
    - total O(m log n)
  - d-Heap (d=m/n)
    - insert, decrease-key $O(\log_{m/n} n)$
    - delete-min $O((m/n) \log_{m/n} n)$
    - total $O(m \log_{m/n} n)$

Minimum Spanning Trees (Forests)

- Given an undirected graph G=(V,E) with each edge e having a weight w(e)
- Find a subgraph T of G of minimum total weight s.t. every pair of vertices connected in G are also connected in T
  - if G is connected then T is a tree otherwise it is a forest

Weighted Undirected Graph



First Greedy Algorithm

- Prim's Algorithm:
  - start at a vertex s
  - add the cheapest edge adjacent to s
  - repeatedly add the cheapest edge that joins the vertices explored so far to the rest of the graph
  - Exactly like Dijsktra's Algorithm but with a different metric

Dijsktra's Algorithm

Dijkstra(**G**,**w**,**s**)
  S←{**s**}
  d[**s**]←**0**
  while **S**≠**V** do
    of all edges **e**=(**u**,**v**) s.t. **v**∉**S** and **u**∈**S** select* one with the minimum value of **d**[**u**]+**w**(**e**)
      S←**S**∪ {**v**}
      d[**v**]←**d**[**u**]+**w**(**e**)
      pred[**v**]←**u**

*For each v∉S maintain d'[v]=minimum value of d[u]+w(e) over all vertices u∈S s.t. e=(u,v) is in of G

Prim's Algorithm

Prim(**G**,**w**,**s**)
  S←{**s**}

  while **S**≠**V** do
    of all edges **e**=(**u**,**v**) s.t. **v**∉**S** and **u**∈**S** select* one with the minimum value of **w**(**e**)
      S←**S**∪ {**v**}

      pred[**v**]←**u**

*For each v∉S maintain small[v]=minimum value of w(e) over all vertices u∈S s.t. e=(u,v) is in of G

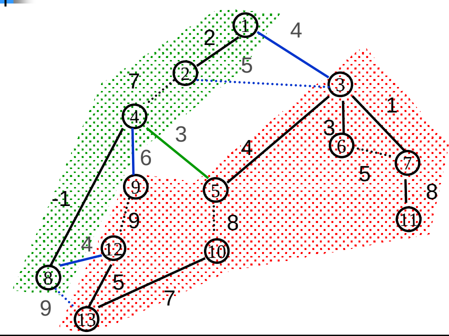
Second Greedy Algorithm

- Kruskal's Algorithm
  - Start with the vertices and no edges
  - Repeatedly add the cheapest edge that joins two different components. i.e. that doesn't create a cycle

Why greed is good

- **Definition:** Given a graph G=(V,E), a **cut** of G is a partition of V into two non-empty pieces, S and V-S
- **Lemma:** For every cut (S,V-S) of G, there is a minimum spanning tree (or forest) containing any **cheapest edge crossing the cut**, i.e. connecting some node in S with some node in V-S.
  - call such an edge **safe**

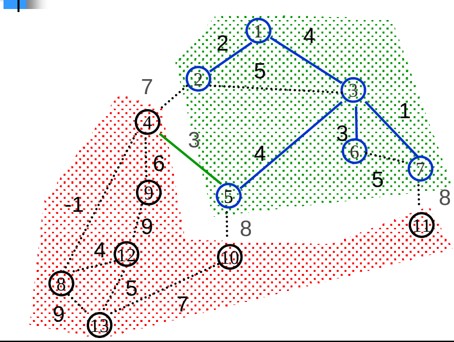
Cuts and Spanning Trees



The greedy algorithms always choose safe edges

- Prim's Algorithm
  - Always chooses cheapest edge from current tree to rest of the graph
  - This is cheapest edge across a cut which has the vertices of that tree on one side.

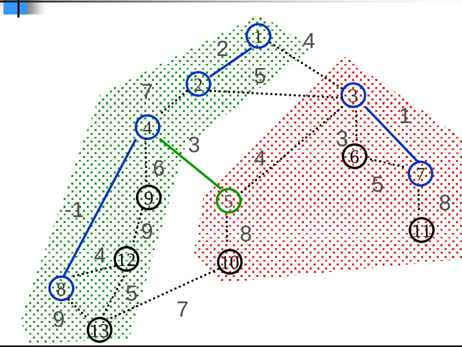
Prim's Algorithm



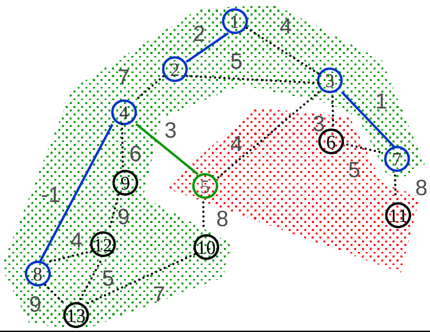
The greedy algorithms always choose safe edges

- Kruskal's Algorithm
  - Always chooses cheapest edge connecting two pieces of the graph that aren't yet connected
  - This is the cheapest edge across any cut which has those two pieces on different sides and doesn't split any current pieces.

Kruskal's Algorithm

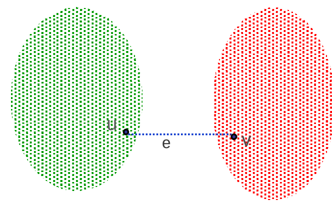


Kruskal's Algorithm



Proof of Lemma: An Exchange Argument

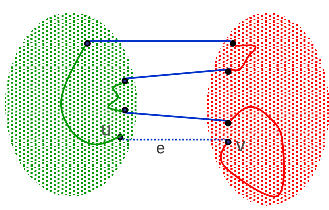
Suppose you have an MST not using cheapest edge e



Endpoints of e, u and v must be connected in T

Proof of Lemma

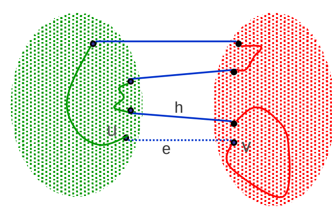
Suppose you have an MST not using cheapest edge e



Endpoints of e, u and v must be connected in T

Proof of Lemma

Suppose you have an MST not using cheapest edge e

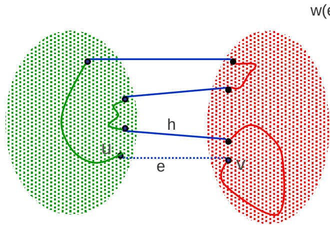


Endpoints of e, u and v must be connected in T

Proof of Lemma

Suppose you have an MST not using cheapest edge e

w(e)≤w(h)

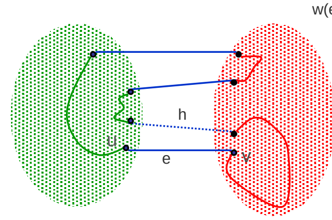


Endpoints of e, u and v must be connected in T

Proof of Lemma

Replacing h by e does not increase weight of T

w(e)≤w(h)



All the same points are connected by the new tree

Kruskal's Algorithm Implementation & Analysis

- First sort the edges by weight O(m log m)
- Go through edges from smallest to largest
  - if endpoints of edge e are currently in different components
    - then add to the graph
    - else skip
- Union-find data structure handles last part
- Total cost of last part: O(m α(n)) where α(n)<< log m
- Overall O(m log n)

85

Union-find disjoint sets data structure

- Maintaining components
  - start with n different components
    - one per vertex
  - find components of the two endpoints of e
    - 2m finds
  - union two components when edge connecting them is added
    - n-1 unions

86

Prim's Algorithm with Priority Queues

- For each vertex u not in tree maintain current cheapest edge from tree to u
  - Store u in priority queue with key = weight of this edge
- Operations:
  - n-1 insertions (each vertex added once)
  - n-1 delete-mins (each vertex deleted once)
    - pick the vertex of smallest key, remove it from the p.q. and add its edge to the graph
  - <m decrease-keys (each edge updates one vertex)

87

Prim's Algorithm with Priority Queues

- Priority queue implementations
  - Array
    - insert O(1), delete-min O(n), decrease-key O(1)
    - total $O(n+n^2+m)=O(n^2)$
  - Heap
    - insert, delete-min, decrease-key all O(log n)
    - total O(m log n)
  - d-Heap (d=m/n)
    - insert, decrease-key $O(\log_{m/n} n)$
    - delete-min $O((m/n) \log_{m/n} n)$
    - total $O(m \log_{m/n} n)$

88

Boruvka's Algorithm (1927)

- A bit like Kruskal's Algorithm
  - Start with **n** components consisting of a single vertex each
  - At each step, each component chooses its cheapest outgoing edge to add to the spanning forest
    - Two components may choose to add the same edge
  - Useful for parallel algorithms since components may be processed (almost) independently

89

Many other minimum spanning tree algorithms, most of them greedy

- Cheriton & Tarjan
  - O(m loglog n) time using a queue of components
- Chazelle
  - O(m α(m) log α(m)) time
    - Incredibly hairy algorithm
- Karger, Klein & Tarjan
  - O(m+n) time randomized algorithm that works most of the time

90

Applications of Minimum Spanning Tree Algorithms

- Minimum cost network design:
  - Build a network to connect all locations $\{v_1,\ldots,v_n\}$
  - Cost of connecting v_i to v_j is $w(v_i,v_j)>0$
  - Choose a collection of links to create that will be as cheap as possible
  - Any minimum cost solution is an MST
    - If there is a solution containing a cycle then we can remove any edge and get a cheaper solution

91

Applications of Minimum Spanning Tree Algorithms

- Maximum Spacing Clustering
  - Given
    - a collection **U** of **n** objects $\{p_1,\ldots,p_n\}$
    - Distance measure $d(p_i,p_j)$ satisfying
      - $d(p_i,p_i)=0$
      - $d(p_i,p_j)>0$ for $i\neq j$
      - $d(p_i,p_j)=d(p_j,p_i)$
    - Positive integer $k\leq n$
  - Find a k-clustering, i.e. partition of U into k clusters $C_1,\ldots,C_k$, such that the **spacing** between the clusters is as large possible where

    spacing = $\min\{d(p_i,p_j)$: p_i and p_j in different clusters$\}$

92

Greedy Algorithm

- Start with n clusters each consisting of a single point
- Repeatedly find the closest pair of points in different clusters under distance d and merge their clusters until only k clusters remain
- Gets the same components as Kruskal's Algorithm does!
  - The sequence of closest pairs is exactly the MST
- Alternatively we could run Kruskal's algorithm once and for any k we could get the maximum spacing k-clustering by deleting the k-1 most expensive edges

93

Proof that this works

- Removing the **k-1** most expensive edges from an MST yields **k** components $C_1,\ldots,C_k$ and the spacing for them is precisely the cost **d*** of the **k-1**st most expensive edge in the tree
- Consider any other **k**-clustering $C'_1,\ldots,C'_k$
  - Since they are different and cover the same set of points there is some pair of points p_i,p_j such that p_i,p_j are in some cluster C_r but p_i, p_j are in different clusters C'_s and C'_t
    - Since $p_i,p_j \in C_r$, p_i and p_j have a path between them all of whose edges have distance at most **d***
    - This path must cross between clusters in the **C'** clustering so the spacing in **C'** is at most **d***

94