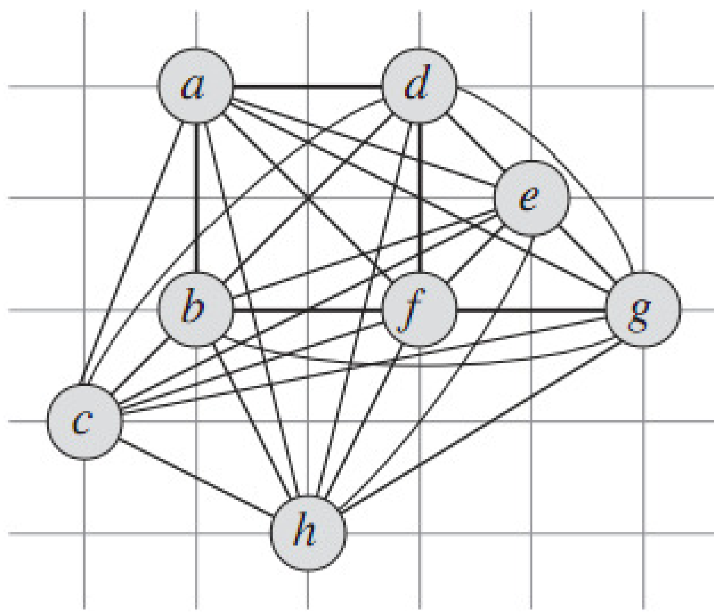
Approximation algorithms 2
TSP, k-Center, Scheduling

CS240 Spring 2022

Rui Fan

Traveling Salesman Problem

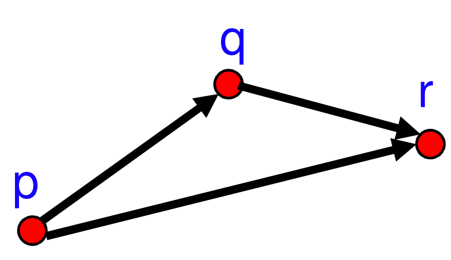
- Input A complete graph with weights on the edges.
- Output A cycle that visits each city once.
- Goal Find a cycle with minimum total weight.



source: CLRS

Metric TSP

- TSP is NP-hard. In fact, it's even NP-hard to approximate when weights can be arbitrary.
- However, TSP is approximable for special types of weights.
- A weighted graph satisfies the triangle inequality if for any 3 vertices p, q, r, we have $d_{pq}+d_{qr} \geq d_{pr}$.
  - I.e., direct path is always no worse than a roundabout path.
  - This is called a metric TSP.
- There is a 1.5-approx algorithm for TSP in graphs with the triangle inequality.
  - Let's look at a simpler 2-approx first.

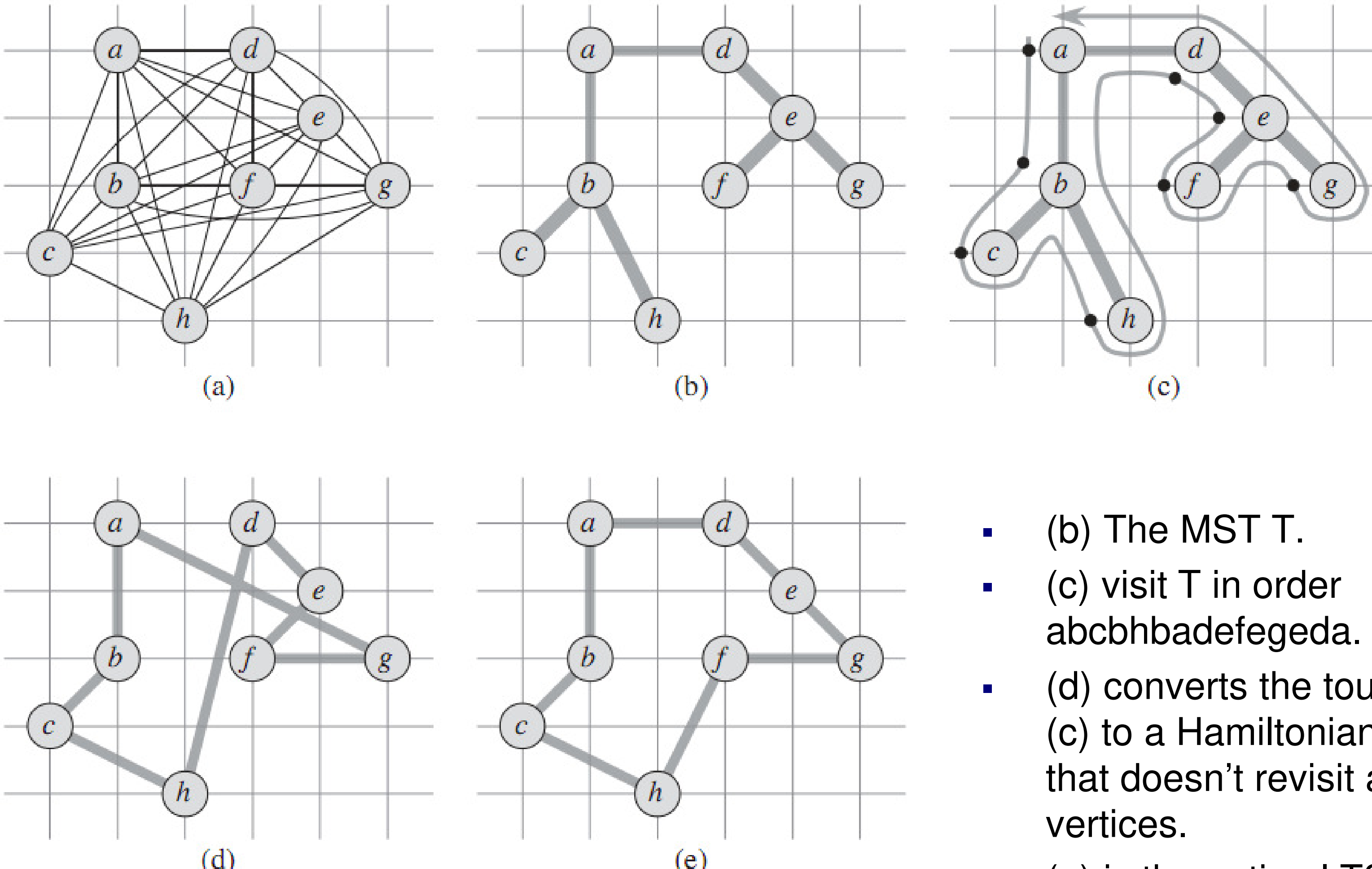


A 2-approximation for TSP

- Construct a minimum spanning tree T on G.
- Use depth-first traversal to visit all the vertices in T, starting from an arbitrary vertex.
- Convert this depth-first traversal T' to a cycle H that doesn't revisit any vertex.
- Return H as the TSP tour.

Example

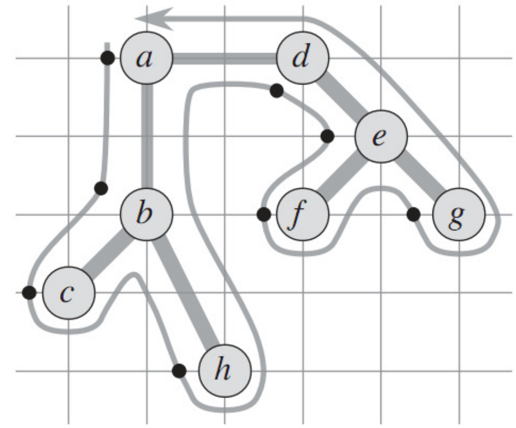
source: CLRS



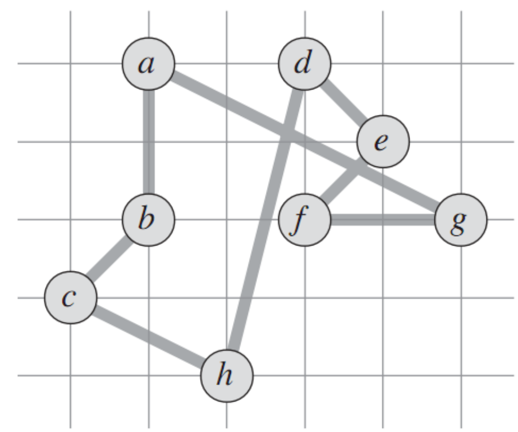
- (b) The MST T.
- (c) visit T in order abcbhbadefegeda.
- (d) converts the tour from (c) to a Hamiltonian cycle, that doesn't revisit any vertices.
- (e) is the optimal TSP.

Making the tour Hamiltonian

- To go from (c) to (d), we need to make a tour T' that revisits vertices into a cycle H that doesn't revisit vertices.
- We use shortcutting.
  - If we revisit a vertex in T', we directly jump to the next vertex in T' we haven't visited.
    - We allow revisiting the first vertex.
  - The sequence of vertices we now visit is H.
  - Ex abcbhbadefegeda → abchdefga.



(c)



(d)

Making the tour Hamiltonian

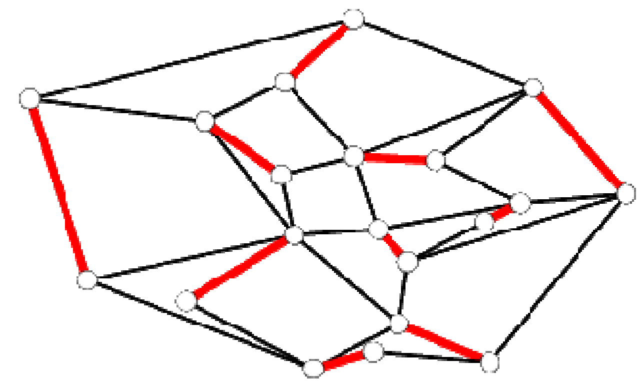
- Lemma If H is the shortcut of T', then c(H)≤c(T').
- Proof We formed H from T' by skipping over some vertices. E.g. we directly went from c to h, skipping over b.
  - But by the triangle inequality, $d_{cb}+d_{bh} \geq d_{ch}$.
    - So shortcutting from c to h didn't increase the distance.
  - The same thing applies to all our shortcuts.
  - So H is no longer than T'.

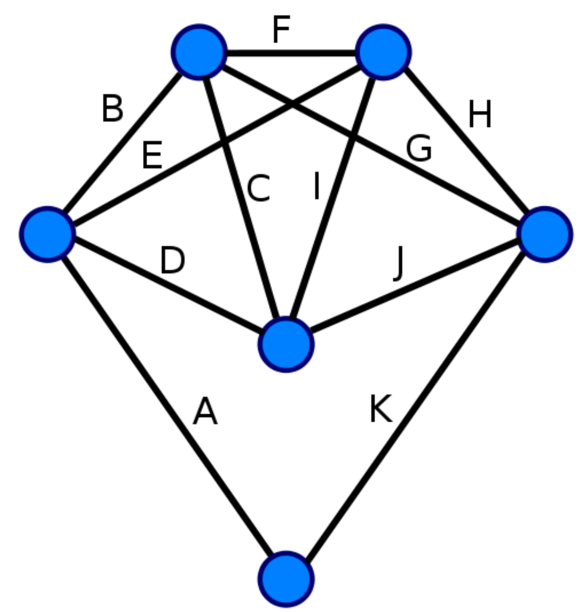
Proof of 2-approximation

- Let H* be an optimum TSP.
- If we delete an edge from H*, we get a spanning tree.
- Since T is an MST, $c(T) \leq c(H^*)$.
- Call the path from the depth-first traversal T'.
  - T' crosses each edge in T twice.
  - So $c(T') = 2\, c(T)$.
- Let H be the outcome of shortcutting T'.
  - H is a Hamiltonian cycle. It visits all the vertices, and ends where it started.
  - $c(H) \leq c(T')$, by the lemma.
  - $c(H) \leq c(T') = 2\, c(T) \leq 2\, c(H^*)$.
- So H is a 2-approximation.

Matchings and Euler cycles

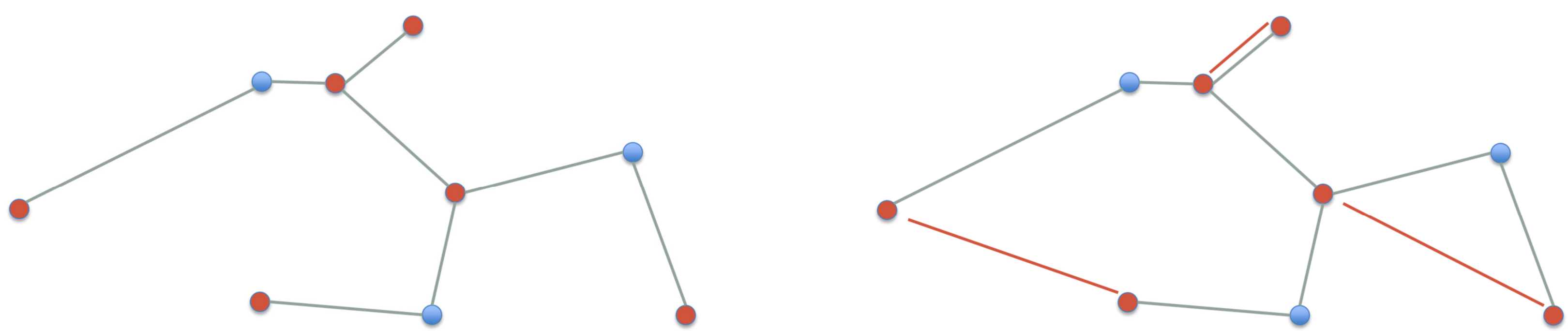
- A matching in a graph is a set of nonintersecting edges.
  - A perfect matching is a matching that includes every vertex.
- An Euler tour of a graph is a path that starts and ends at the same vertex, and visits every edge once.
  - Hamiltonian tour visits every vertex once.
- Thm (Euler) A graph has an Euler tour if and only if all vertices have even degree.
- Note how deciding if graph has Euler tour is trivial, but deciding if it has Hamiltonian tour is NPC!





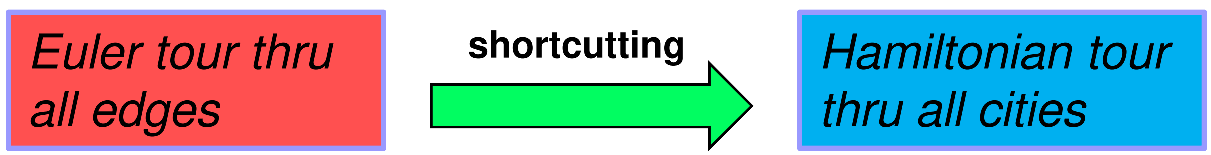
Christofides 3/2-approx algorithm

- A 3/2-approximation for TSP with triangle inequality.
- Construct a minimum spanning tree T on G.
- Find the set V' of odd degree vertices in T.
- Construct a minimum cost perfect matching M on V'.
- Add M to T to obtain T'.
- Find an Euler tour T'' in T'.
- Shortcut T'' to obtain a Hamiltonian cycle H. Output as the TSP.



Why Christofides works well

- In the 2-approx, we found a TSP by “doubling” the MST to an Euler tour, then shortcutting.
  - We need to start with Euler tour before shortcutting to ensure we visit all cities.



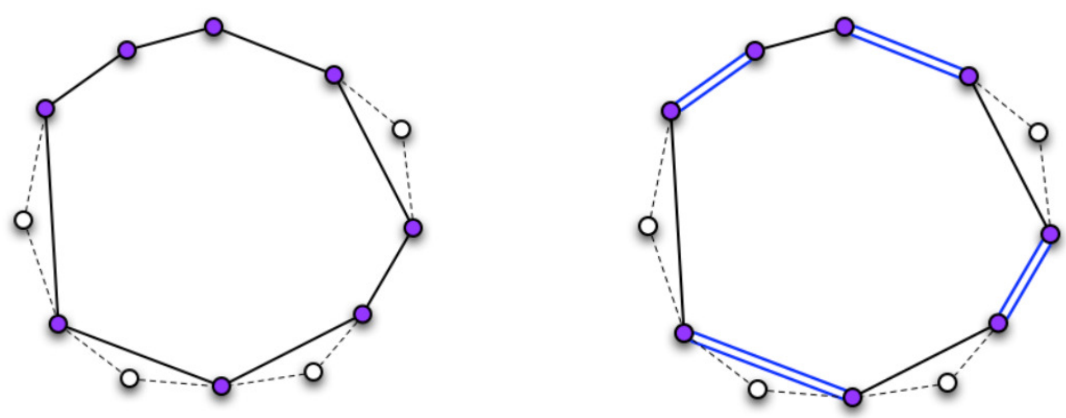
- Key to Christofides is to find a shorter Euler tour, without doubling the MST.
  - A graph with only even degree vertices always has Euler tour.
  - So we want to modify the MST to have all even degrees, by adding a matching.

Proof of correctness

- Lemma T' has an Euler tour.
- Proof There are an even number of vertices in V', because the total degree of T is even.
  - Since G is a complete graph and |V'| is even, there's a perfect matching on V'.
    - The min cost perfect matching can be found in $O(n^2)$ time using the blossom algorithm.
  - The degree of every node in M is odd. Since V' are the odd degree nodes in T, adding M to T makes all nodes in T' have even degree.
  - T' has Euler tour by Euler's theorem.

Proof of correctness

- Lemma Let H^* be an optimal TSP on G, and let m be the cost of M. Then $m \leq c(H^*)/2$.
- Proof Let H' be the optimal TSP on V'.
  - $c(H') \leq c(H^*)$ because H' is an optimal TSP on fewer vertices.
  - H' is a cycle on V', so it consists of two matchings on V'. The cheaper one has cost $m' \leq c(H')/2 \leq c(H^*)/2$.
  - $m \leq m'$ because M has min cost.



http://www.mpi-inf.mpg.de/departments/d1/teaching/ss12/AdvancedGraphAlgorithms/Slides06.pdf

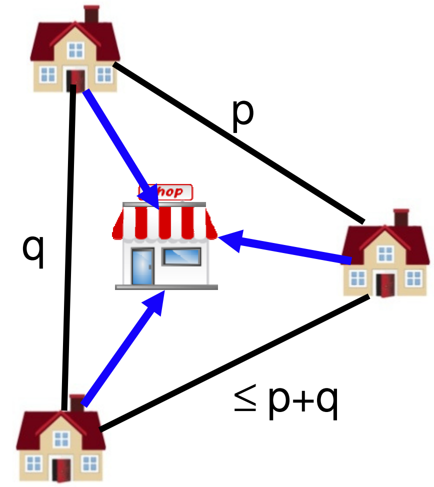
Proof of 3/2-approximation

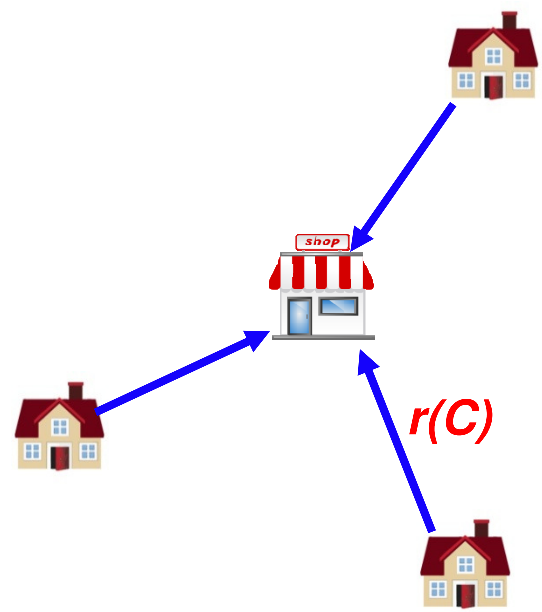
- Thm Let H be the TSP output by Christofides and let H* be an optimal TSP. Then $c(H) \leq 3/2 * c(H^*)$.
- Proof
  - $c(T) \leq c(H^*)$ because T is an MST.
  - $c(T') = c(M) + c(T) \leq c(H^*)/2 + c(H^*) = 3/2 * c(H^*)$.
  - $c(H) \leq c(T')$ because H is the shortcut of T'.

- *Construct a minimum spanning tree T on G.*
- *Find set V' of odd-degree vertices in T.*
- *Construct a minimum cost perfect matching M on V'.*
- *Add M to T to obtain T'.*
- *Shortcut T' to obtain a Hamiltonian cycle. Output as the TSP.*

k-Center problem

- Given a city with n sites, we want to build k centers to serve them.
  - Let S be set of sites, C be set of centers.
- Each site uses the center closest to it.
  - Distance of site s from the nearest center is $d(s,C) = \min_{c\in C} d(s,c)$.
- Goal is to make sure no site is too far from its center.
  - We want to minimize the max distance that any site is from its closest center.
    - Minimize $r(C)=\max_{s\in S} \min_{c\in C} d(s,c)$.
  - C is called a cover of S, and r is called C's radius.
  - Where should we put centers to minimize the radius?
- Assume distances satisfy triangle inequality.





Gonzalez's algorithm

- k-Center is NP-complete.
- We'll give a simple 2-approximation for it.
- Idea Say there's one site that's farthest away from all centers. Then it makes the radius large. We'll put a center at that site, to reduce the radius.
  - Note we allow putting center at same location as site.

Gonzalez's algorithm

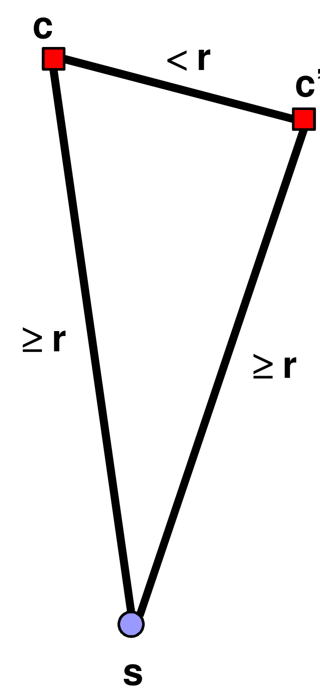
- C is set of centers, initially empty.

- repeat k times
  - choose site s with maximum d(s,C)
  - add s to C
- return C

- Note The centers are located at the sites.

Proof of correctness

- Let C be the algorithm's output, and r be C's radius.
  - $r = \max_{s\in S} \min_{c\in C} d(s,c)$
- Lemma 1 For any $c,c' \in C$, $d(c,c') \geq r$.
- Proof Since r is the radius, there exists a point $s \in S$ at distance $\geq r$ from all the centers.
  - If there's no such s, then C's radius < r.
  - So s is distance $\geq r$ from c and c'.
  - Suppose WLOG c' is added to C after c.
  - If $d(c,c')<r$, then algorithm would add s to C instead of c', since s is farther.

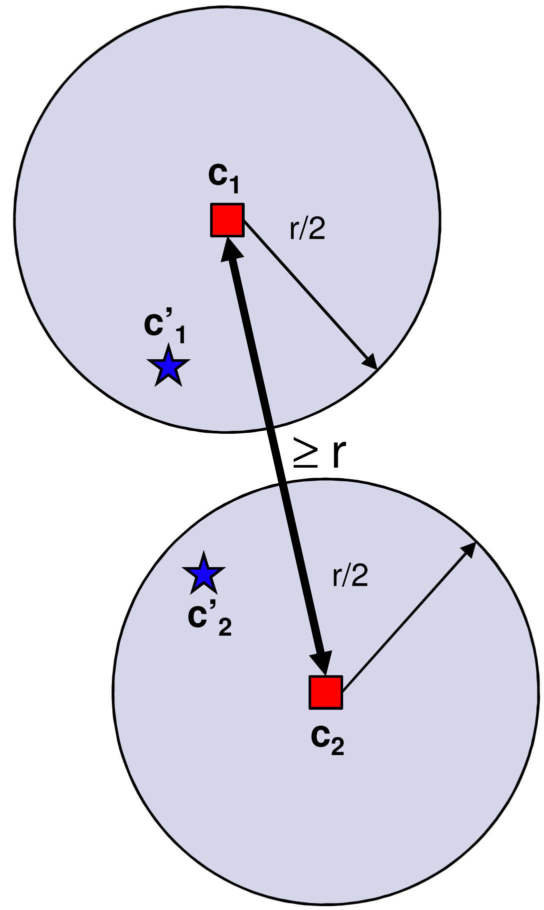


Proof of correctness

- Cor There exist k+1 points mutually at distance $\geq$ r from each other.
  - By the lemma, the k centers are mutually $\geq$ r distance apart.
  - Also, there's an $s \in S$ at distance $\geq$ r from all the centers.
    - Otherwise C's covering radius is < r.
  - So the k centers plus s are the k+1 points.
- Call these k+1 points D.

Proof of correctness

- Let C^* be an optimal cover with radius r^*.
- Lemma 2 Suppose $r > 2r^*$. Then for every $c \in D$, there exists a corresponding $c' \in C^*$. Furthermore, all these c' are unique.
- Proof Draw a circle of radius $r/2$ around each $c \in D$.
  - There must be a $c' \in C^*$ inside the circle, because
    - c is at most distance r^* away from its nearest center, since r^* is C^*'s radius.
    - $r/2 > r^*$.
  - Given $c_1, c_2 \in D$, let $c'_1, c'_2 \in C^*$ be inside c_1 and c_2's circle, resp.
  - c_1 and c_2's circles don't touch, because $d(c_1, c_2) \geq r$.
  - So $c'_1 \neq c'_2$.

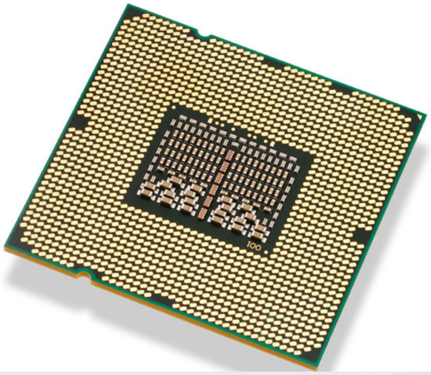


Proof of correctness

- Thm Let C be the output of Gonzalez's algorithm, and let C^* be an optimal k-center. Then $r(C) \leq 2r(C^*)$.
- Proof By Lemma 2, if $r(C)>2r(C^*)$, then for every $c \in D$, there is a unique $c' \in C^*$.
  - But there are k+1 points in D, by the corollary.
  - So there are k+1 points in C^*. This is a contradiction because C^* is a k-center.

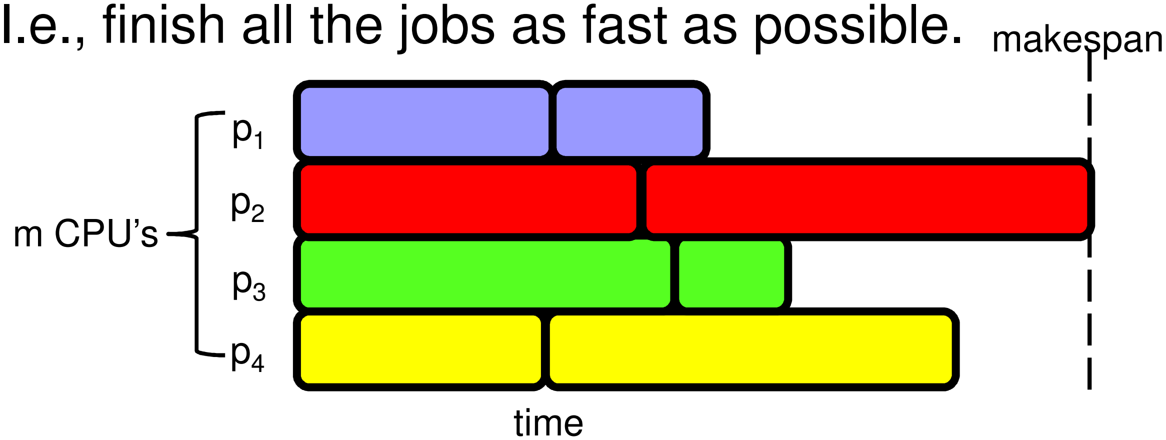
Parallel computing and scheduling

- Computers today are parallel.
  - Multiple processors in a system.
  - Multiple tasks for the processors to run.
- Multiprocessor scheduling is the problem of deciding which tasks to run on which processors at what time.
- Many possible objectives.
  - Throughput, fairness, energy usage.
  - Latency, i.e. finishing all jobs as fast as possible.



Makespan scheduling

- n independent jobs.
  - Jobs have different sizes, i.e. time needed to perform job.
  - Jobs can be done in any order.
  - Any job can be done on any machine.
- m processors.
  - All have the same speed.
  - Each processors can do one job at a time.
- Assign the jobs to the processors.
- Makespan is when the last processor finishes all its jobs.
- Minimize the makespan.
  - I.e., finish all the jobs as fast as possible.



Minimizing makespan is NPC

- The decision version of scheduling is obviously in NP.
- SUBSET-SUM: given a set of numbers S and target t, is there a subset of S summing to t?
  - Ex S={1,3,8,9}. t=9, yes. t=14, no.
  - This is NP-complete. We reduce SUBSET-SUM to scheduling.
- Let (S,t) be an instance of SUBSET-SUM.
  - Let s be sum of all elements in S.
- Make a set of jobs $J = S \cup \{s-2t\}$, and schedule them on 2 processors.

Minimizing makespan is NPC

- Claim If some subset of S sums to t, then min makespan is s-t.
- Proof Say S'⊆S sums to t. Schedule the jobs in S' and job s-2t on processor 1. So proc 1 finishes at time t+s-2t=s-t. Proc 2 does the jobs in S-S', so it finishes at time s-t as well.
- Claim If the min makespan is s-t, there exists a subset of S that sums to t.
- Proof Suppose WLOG proc 1 does the s-2t job. Since makespan is s-t, the other jobs proc 1 does must have total size s-t-(s-2t)=t.
- So (S,t) is yes instance of SUBSET-SUM iff makespan = s-t.
  - So SUBSET-SUM $\leq_p$ scheduling, and scheduling is NP-complete.

Graham's list scheduling

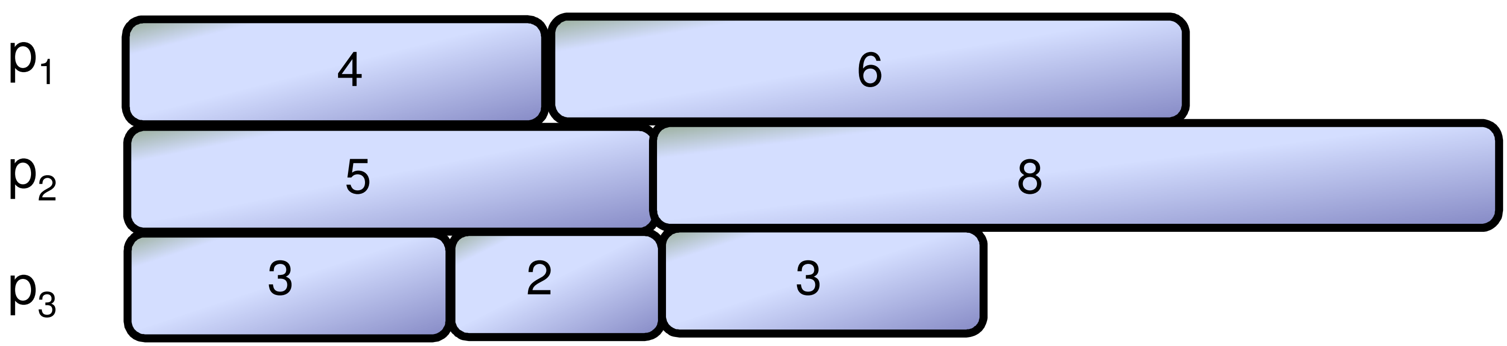
- Since scheduling is NPC, it's unlikely we can find the min makespan in polytime.
- List scheduling is a simple greedy algorithm.
  - Finds a schedule with makespan at most twice the minimum.
  - A 2-approximation.
- If there are n tasks and m processors, list scheduling only takes O(n log n) time.
  - Compare this to n! C(n+m-1, m-1) time to try all possible schedules and pick the best.

Graham's list scheduling

- List the jobs in any order.
- As long as there are unfinished jobs.
  - If any processor doesn't have a job now, give it the next job in the list.

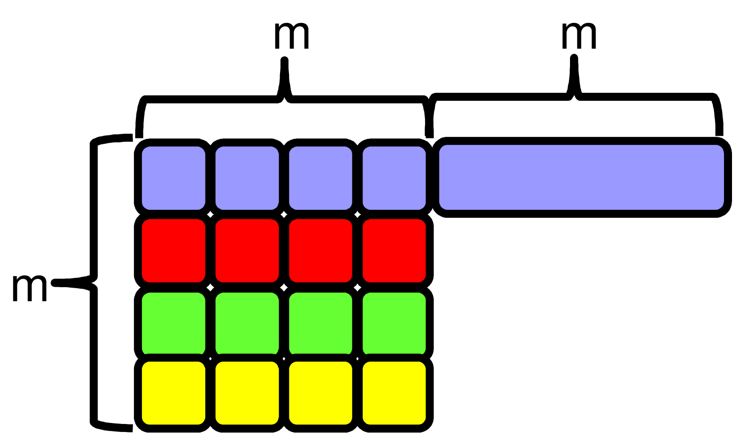
Example

- 3 processors. The jobs have length 2, 3, 3, 4, 5, 6, 8.
- List them in any order. Say 4, 5, 3, 2, 6, 8, 3.
- Initially, no proc has a job. Give first 3 jobs to the 3 procs.
- At time 3, proc 3 is done. Give it next job in list, 2.
- At time 4, proc 2 is done. Give it next job in list, 6.
- At time 5, both 1, 3 are done. Give them next jobs in list, 8,3.
- Everybody finishes by time 13.
  - The makespan of this schedule is 13.

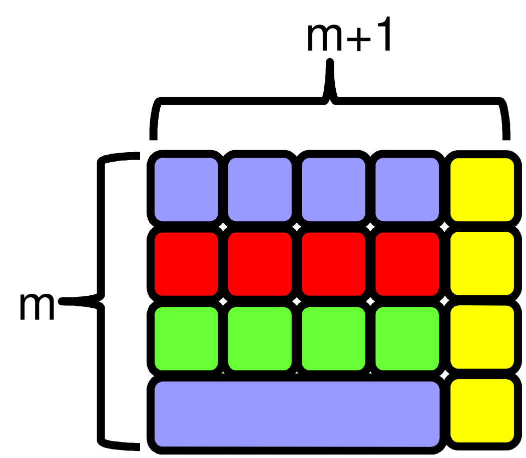


The worst case for LS

- How badly can list scheduling do compared to optimal?
- Say there are m^2 jobs with length 1, and one job with length m.
  - Suppose they're listed in the order 1,1,1,...,1,m.
  - LS has makespan 2m. Optimal makespan is m+1.
  - makespan(LS) / makespan(opt) = 2m/(m+1) ≈ 2.
- This is worst possible case for list scheduling.



makespan(LS) = 2m

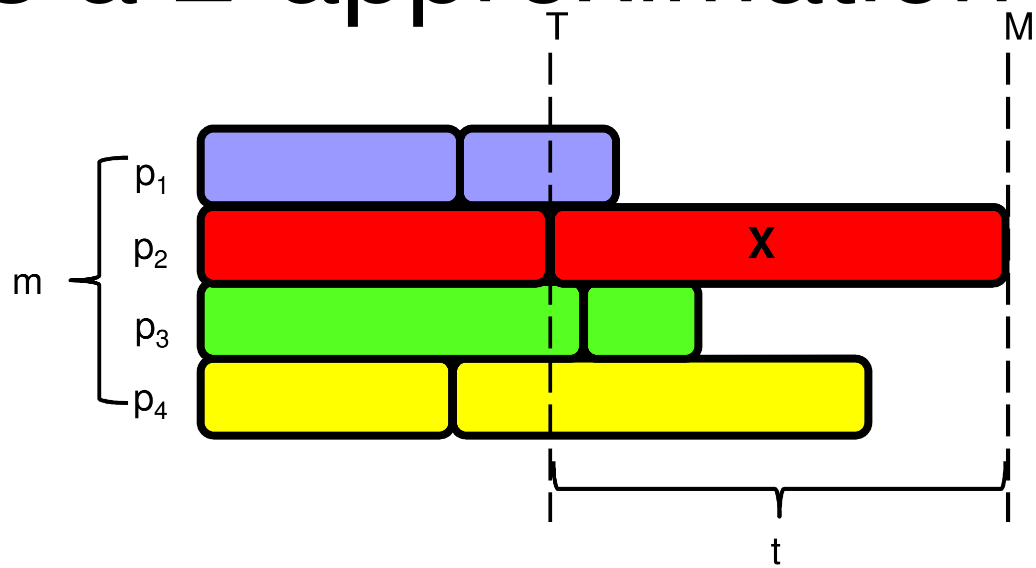


makespan(opt) = m+1

LS is a 2-approximation

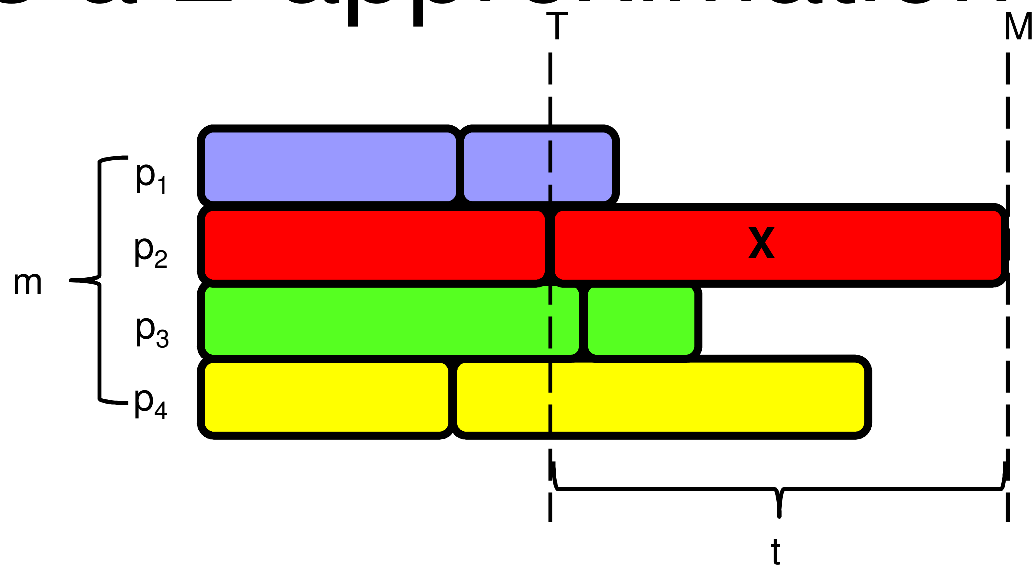
- Next, we prove LS always gives a schedule at most twice the optimal.
- Suppose LS gives makespan of M.
- Let the optimal schedule have makespan M*.
- We prove that $M \leq 2M^*$.

LS is a 2-approximation



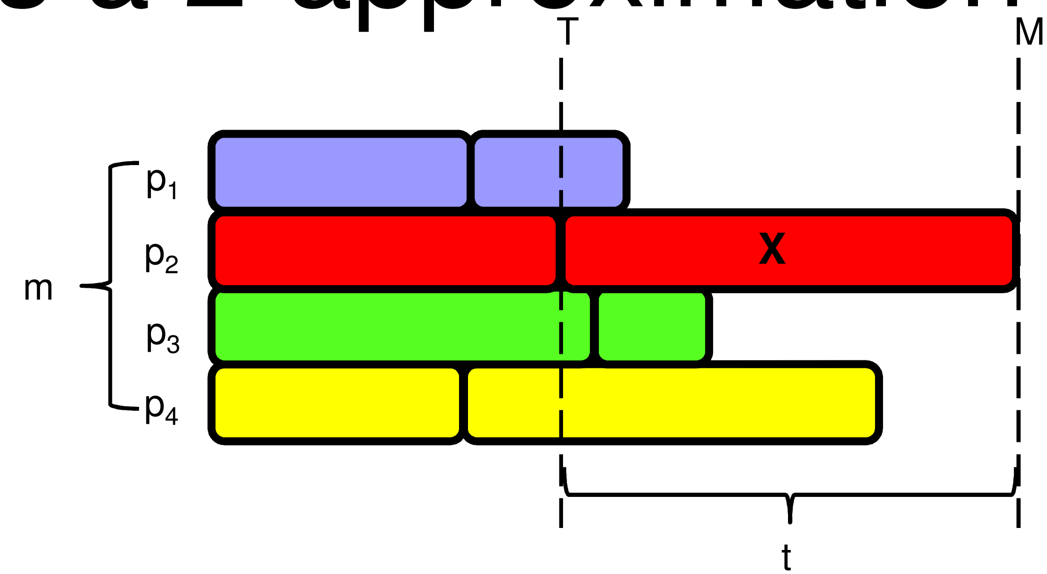
- The picture above is the schedule produced by list scheduling.
- Consider task X that finishes last.
  - Say X starts at time T, and has length t.
- Claim 1 $M^* \geq t$.
  - In any schedule, X has to run on some process.
  - Since X takes t time, every schedule, including the opt, takes $\geq$ t time.

LS is a 2-approximation



- Claim 2 $M^* \geq T$.
  - Up to time T, no processor is ever idle.
    - Up to T, there's always some unfinished job.
    - As soon as a processor finishes one job, it's assigned another one.
  - So at time T, each processor completed T units of work.
  - So total amount of work in all the jobs is $\geq mT$.
  - In the opt schedule, m processors complete at most m units of work per time unit.
  - So length of opt schedule is $\geq$ (total work)/m $\geq mT/m = T$.

LS is a 2-approximation



- From Claims 1 and 2, we have $M^* \geq t$ and $M^* \geq T$.
- So $M^* \geq \max(T,t)$.
- $M = T + t$, because X is last job to finish.
- So $M/M^* \leq (T+t)/\max(T,t) \leq 2$.

LPT scheduling

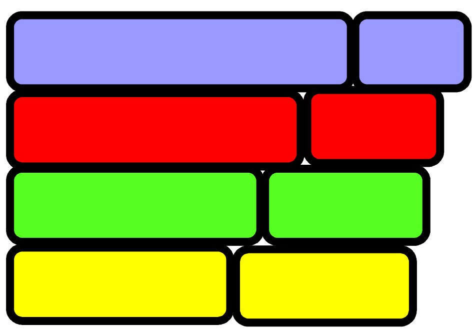
- Worst case for LS occurred when longest job was scheduled last.
  - Large jobs are “dangerous” at end.
- Let’s try to schedule longest jobs first.
- Longest processing time (LPT) schedule is just like list scheduling, except it first sorts tasks by nonincreasing order of size.
- Ex For three processors and tasks with sizes 2, 3, 3, 4, 5, 6, 8, LPT first sorts the jobs as 8,6,5,4,3,3,2. Then it assigns p_1 tasks 8,3, p_2 tasks 6,3, p_3 tasks 5,4,2, for a makespan of 11.
- LPT has an approximation ratio of 4/3.

LPT is a 4/3-approximation

- Thm Suppose the optimal makespan is M*, and LPT produces a schedule with makespan M. Then $M \leq 4/3\ M^*$.
- Let X be the last job to finish. Assume it starts at time T and has size t.
- Assume WLOG that X is the last job to start.
  - If not, then say Y starts after T.
  - Y finishes before T+t. So we can remove Y without increasing the makespan.
- Cor 1 X is the smallest job.
  - X is the last job to start, so due to LPT scheduling it's the smallest.

LPT is a 4/3-approximation

- Claim 1 LPT's makespan = $T+t \leq M^*+t$.
  - As in LS, no processor is idle up to time T, so $M^* \geq T$.
- Case 1 $t \leq M^*/3$.
  - Then LPT's makespan $\leq M^* + t \leq M^* + M^*/3 = 4/3\ M^*$.
- Case 2 $t > M^*/3$.
  - Since X is the smallest task, all tasks have size $> M^*/3$.
  - So the optimal schedule has at most 2 tasks per processor. So $n \leq 2m$.
  - If $1 \leq n \leq m$, then LPT and optimal schedule both put one task per processor.
  - If $m < n \leq 2m$, then optimal schedule is to put tasks in nonincreasing order on processors 1,...,m, then on m,...,1.
    - LPT also schedules tasks this way, so it's optimal.



LS vs LPT

- LPT gives better approx ratio, has same running time. Why bother with LS?
- LS is online.
  - Imagine the jobs are coming one by one.
    - LS just puts them on any idle computer.
- LPT is offline
  - It needs to know all the jobs that will ever arrive, in order to sort them.
- In a realistic parallel computation, you get jobs on the fly.
  - Online is more realistic.
  - LS is usually more useful.