Exact Algorithms for NP-hard problems

Advanced Algorithms: Part 2, Lecture 2

Today

- Revisit of search trees: summary & homework
- Recapture of Dynamic Programming
- Dynamic Programming for NP-hard problems
  - Traveling salesperson problem
  - Scheduling with precedence constraints
  - Circular arc coloring

Mathijs de Weerdt

1-Slide Summary on Search Trees

Search tree

- root represents the complete problem
- children are smaller subproblems: alternatives for single decision (mutually exclusive, all need to be investigated)
- the smaller subcases the better (using worst case analysis)!
  - we may have different types of branches at various places in the tree (first do easier cases, e.g., start with vertices with a single neighbor)
- expressed as recursive algorithm
- resolve recurrence of worst case by assuming exponential runtime

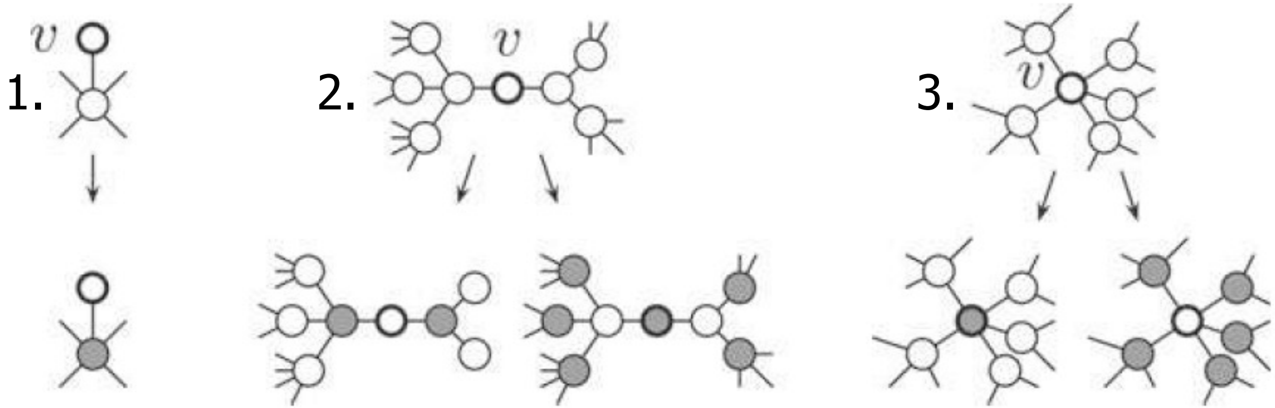
"Examples":

- **independent set**: special cases of 0, 1 and 2 neighbors
- **3 SAT**: 3 branches: $L_1=1$, $L_1=0$ and $L_2=1$, $L_1=L_2=0$ and $L_3=1$
- **vertex cover**: $O^*(2^k)$ and $O^*(1.47^k)$: special cases of 0 and 1; case with degree 2, but worst case is with degree 3

Fixed parameter tractable if runtime bounded by $O(f(k) \cdot p(n))$

TU Delft

Bounded Search Trees: Improving Vertex Cover



Analyze the worst case

1. only one subproblem of size k-1 (thus linear in k)
2. two subproblems: one of size k-2 and one of size at most k-3
3. two subproblems: one of size k-1 and one of size at most k-3

So, case 3 is the worst case…

Recurrence relation describing the run time T(k)

$T(k) \leq T(k-1) + T(k-3) + O(n+m)$ leads to $O^*(1.47^k)$

Fixed parameter tractable

Def. A problem of size n is *fixed parameter tractable (FPT) with respect to parameter k* if it can be solved in f(k) · p(n) time, where

- f is a (usually exponential) function depending only on the parameter k
- p is a polynomial function

To distinguish between behavior:

- $O(f(k) \cdot p(n))$
- $\Omega(n^{f(k)})$

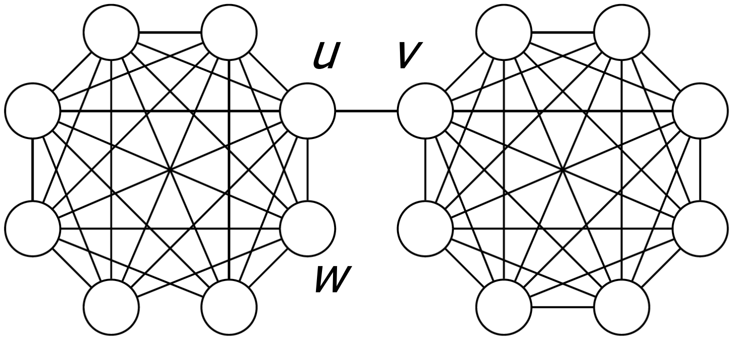
Parameterized complexity was first described by Downey & Fellows (1999).

Q. Given the previous rules, what is f(k) for vertex cover FPT?

A. $f(k)=1.47^k$

p(n) is the time we need select an edge, and an upper bound on the time for preprocessing

Homework: Cluster editing



Q. Can this graph be changed into a *disjoint union of cliques* in at most k=2 edits?

(edit = add or remove an edge)

A. Yes, remove edge in middle and add in the bottom left.

Machine "intelligence"

Classification problem

- Edges represent similarity
- We aim to find a pattern: classes of similar items
- But data is incomplete…

Idea. Use search tree algorithm:
There should be no vertices u, v, w where {u,v} and {u,w} are edges but {v,w} is not.

Homework: Cluster editing

To prove: A graph G = (V,E) is a disjoint union of cliques *if and only if* there are no three distinct vertices u, v, w ∈ V with {u,v} ∈ E and {u,w} ∈ E, but {v,w} ∉ E.

Proof: ⇒

By contradication:

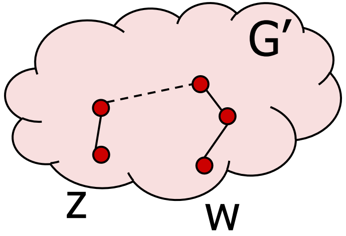
- suppose G is disjoint union of cliques
- suppose we have such three vertices
- observe:
  1. these are not part of a clique
  2. these are not disjoint
- contradiction with G being disjoint union of cliques
- there can thus not be such three vertices

Homework: Cluster editing

To prove: A graph G = (V,E) is a disjoint union of cliques *if and only if* there are no three distinct vertices u, v, w ∈ V with {u,v} ∈ E and {u,w} ∈ E, but {v,w} ∉ E.

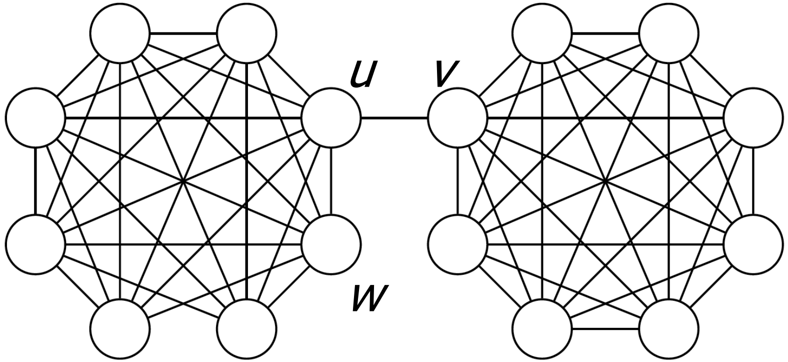
Proof: ⇐

By contraposition (we show that not left implies not right):



- suppose G is not a disjoint union of cliques
- there must be a connected subgraph G′ with at least three vertices that is not a clique (subgraphs of size one & two are cliques)
- observe:
  1. there must be two vertices in G′ that are not directly connected; call these z and w
  2. consider the shortest path from z to w
  3. let u be last vertex before w and v be second-to-last
  4. then {u,v} ∈ E and {u,w} ∈ E, but {v,w} ∉ E

Homework: Cluster editing



Search tree algorithm

Q. What to branch on and what are the sub-cases?
A. Branch on u,v,w where {u,v} and {u,w} are edges but {v,w} is not.
Options:
1. add {v,w}
2. remove {u,v}
3. remove {u,w}

Stop if no edits allowed anymore. (At most k.)

Q. Runtime?
A. Tree of depth k with branching factor 3. So $O^*(3^k)$.

TU Delft

Recapture Dynamic programming

Chapter 6 in Kleinberg & Tardos

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has limit of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Q. What is the maximum value here?
A. { 3, 4 } attains 40

weight limit W = 11

Item i	Value v_i	Weight w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

TU Delft

Dynamic Programming: Adding a New Variable

Recursively define value of optimal solution:
Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best set out of { 1, 2, …, i-1 } using weight limit w
- Case 2: OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best set out of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\left\{ OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i) \right\} & \text{otherwise} \end{cases}$$

base case:
no items left

Q. What is the runtime if implemented as a search tree?
A. $O^*(2^n)$

Knapsack Algorithm: Bottom-Up

W + 1

OPT(i,w):

i:	w:	0	1	2	3	4	5	6	7	8	9	10	11
0	ϕ												
1	{ 1 }												
2	{ 1, 2 }												
3	{ 1, 2, 3 }												
4	{ 1, 2, 3, 4 }												
5	{ 1, 2, 3, 4, 5 }												

n + 1

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i) \,\} & \text{otherwise} \end{cases}$$

Delft

Knapsack Algorithm: Bottom-Up

W + 1 →

OPT(i,w):

i:	w:	0	1	2	3	4	5	6	7	8	9	10	11
0	ϕ	0	0	0	0	0	0	0	0	0	0	0	0
1	{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
2	{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
3	{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
4	{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
5	{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

n + 1

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w),\ \ v_i + OPT(i-1, w - w_i) \} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Compute value of optimal solution iteratively.
Knapsack. Fill up an n-by-W array.

```
Input: n, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Q. What is the runtime?
A. Θ(n W).

Dynamic Programming Summary (Prerequisite)

Recipe.

1. Characterize structure of problem (like the search trees).
2. Recursively define *value* of optimal solution.
3. Compute and store *values* of optimal solution iteratively.
4. Construct optimal solution itself from computed information.

Dynamic programming techniques.

- Binary choice: weighted interval scheduling
- Multi-way choice: segmented least squares.
- Adding a new variable: knapsack.
- Dynamic programming over intervals (more subproblems): RNA secondary structure.

Dynamic programming for NP-hard problems

Dynamic programming (DP) versus search trees

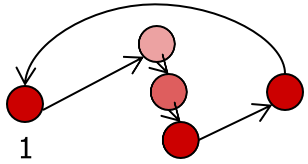
- Both start from a *recursive* definition of the solution
- Search tree: *preventing* common subproblems
- DP: about *reusing* solutions to subproblems

Dynamic programming for NP-hard problems

Three strongly NP-hard problems:

- Traveling salesperson problem
- Scheduling with precedence constraints
- Circular Arc Coloring

Traveling Salesperson Problem

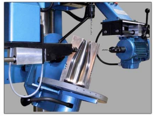


Given
- n cities with distances d(i,j) (no assumptions e.g. on triangle inequality)

Find
- the shortest path from city 1 through all cities and back to 1

Q. What is the runtime of a trivial algorithm?

A. Try each sequence, so $O^*(n!)$

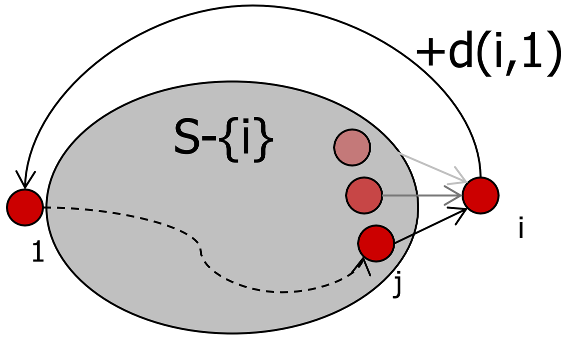


Solve using dynamic programming (or search trees):
- what to branch on?
- what are the sub-cases?

Idea.
- First solve problem of shortest path through all cities ending in i
- Branch on the previous city j (before i):
  - Compute shortest path from 1 through a *subset S* ending in a city j.

i.e., the cities not in the rest of the path (that starts from j)

Traveling Salesperson Problem



Subproblems "via j":
for each j∈S-{i}, find shortest path through S-{i} ending in j

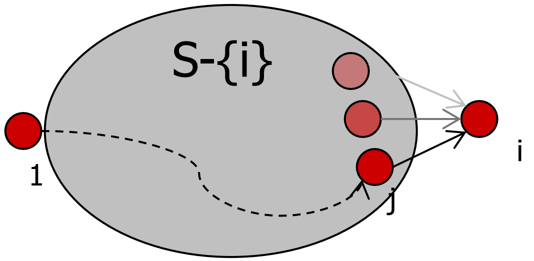
Q. If S=all cities, how to complete the tour and find the min. total length?

Complete the tour: do this for every i ∈ S-{1} and select the minimum length shortest path including returning to 1.

Traveling Salesperson Problem

Let OPT[S;i] denote the shortest path from 1 through all of S ending in i (where S includes i), d(i,j) denotes distance between cities i and j

Express recursively in its subproblems



Q. How to express OPT[S;i], i.e., shortest path from 1 ending in i, in subproblems? (3 min)

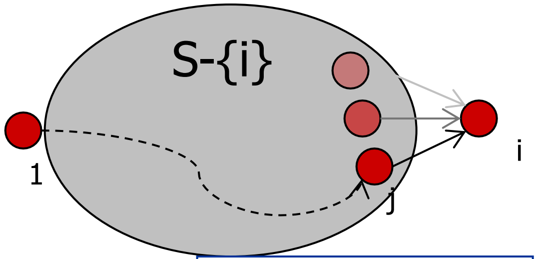
1. Compute shortest path ending in j (recursively) then get from j to i immediately, via j (so j is second-to-last) cost: OPT[S-{i};j] + d(j,i)

2. Take minimum over all possible j:
$$\text{OPT}[S;i] = \min_{j \in S-\{i\}} \{ \text{OPT}[S-\{i\};j] + d(j,i) \}$$

Traveling Salesperson Problem

$OPT[S;i] = \min_{j \in S-\{i\}} \{ OPT[S-\{i\};j] + d(j,i) \}$

(OPT[S;i] = shortest path from 1 through all of S ending in i)



Compute and store iteratively bottom-up

Q. What is the base case?
(shortest path from 1 through S = {i}, ending in i)
$OPT[\{i\};i] = d(1,i)$; we have this for every i

Q. In which order to compute the subproblems?
A. "bottom-up" = smallest subsets first

Q. How to solve TSP using the stored solutions?
A. optimal travel length for complete TSP is then given by
$\min_{i \in \{2,\ldots,n\}} \{ OPT[\{2,\ldots,n\};i] + d(i,1) \}$

PS: A similar definition of OPT[S;i] exists where S never includes i.

Traveling Salesperson Problem

```
DP4TSP(d,{1,2,…,n}) {
   foreach (city i) {
      M[{i};i] = d(1,i)
   }
   foreach (…) {
      foreach (city i in S) {
        M[S;i] = min_{j∈ S-{i}} { M[S-{i};j] + d(j,i) }
   }}
   return min_{i∈{2,…,n}} { M[{2,…,n};i] + d(i,1) }
}
```

Q. What should be on the dots (…)?

A. `j from {1,2,3,…,n} with j≥i`

B. `j from {1,2,3,…,n} with i≥j`

C. `subset S of {2,3,…,n}, increasing in size`

D. `subset S of {2,3,…,n}, increasing in last city number`

Traveling Salesperson Problem

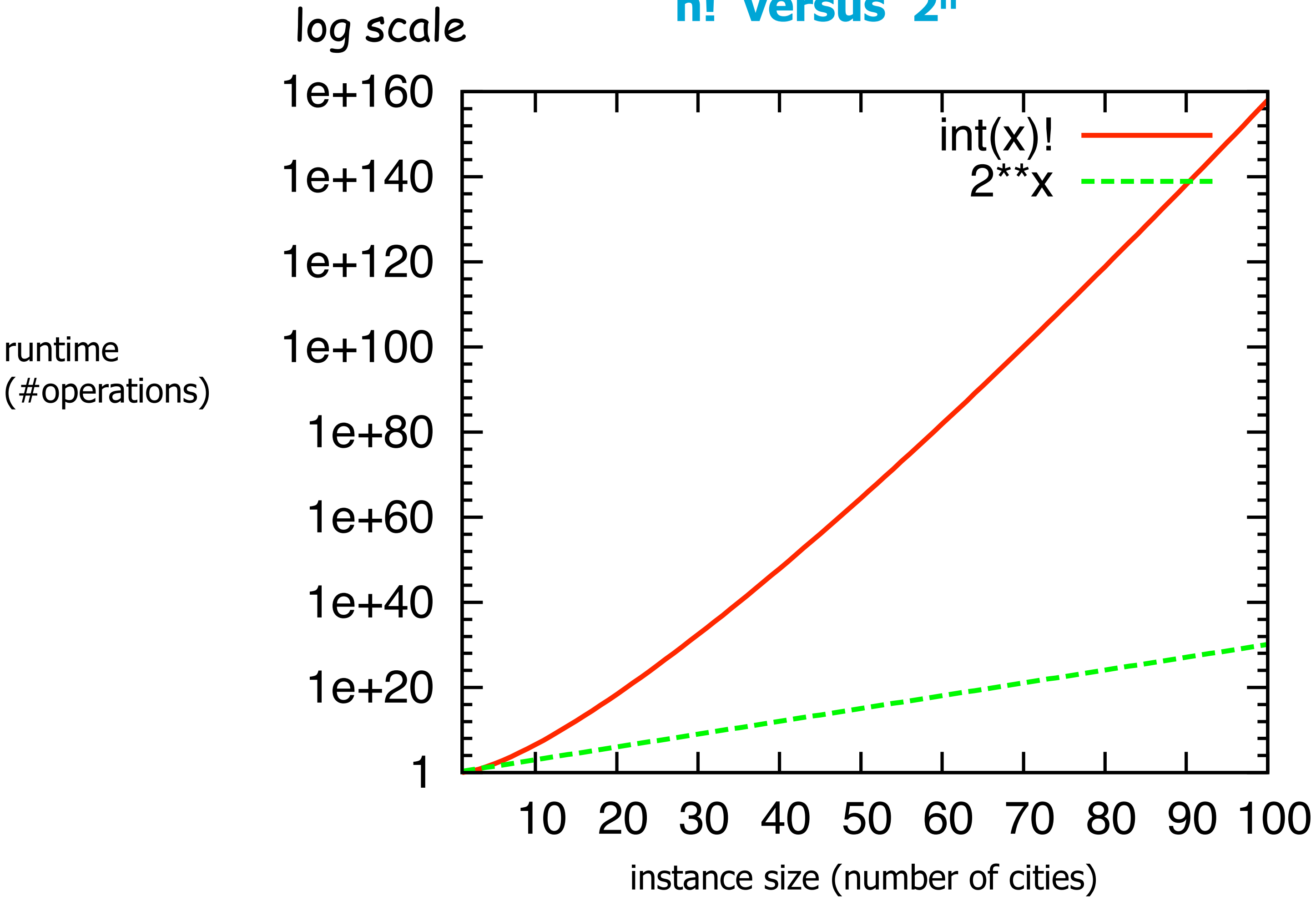
```
DP4TSP(d,{1,2,…,n}) {
   foreach (city i) {
      M[{i};i] = d(1,i)
   }
   foreach (subset S of {2,3,…,n} in increasing size) {
      foreach (city i in S) {
         M[S;i] = min_{j∈ S-{i}} { M[S-{i};j] + d(j,i) }
      }
   }
   return min_{i∈{2,…,n}} { M[{2,…,n};i] + d(i,1) }
}
```

Q. What is a tight bound on space and runtime of this algorithm?

A. array is 2^n for all subsets S, times n for all i, so space is $n \cdot 2^n$
filling it takes time O(n), so $O(n^2 \cdot 2^n) = O^*(2^n)$

NB: this is the best known exact algorithm for general TSP (2003) (non-Euclidean)

n! versus 2^n



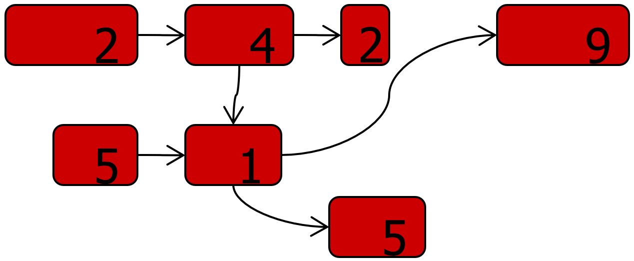
TU Delft

Scheduling with precedence constraints

Given
- 1-machine, set J of n jobs, each with a length p_j and a weight w_j
- precedence constraints (partial order), i.e. i precedes j iff $i \rightarrow j$

Find
- non-preemptive schedule with completion times C_j for each job j
- obeying precedence constraints, and with
- minimum sum of weighted completion times $\Sigma_j^n w_j C_j$

Schedule:
- start time s_j for every job j
- $C_j = s_j + p_j$
- no jobs $i \neq j$ with $s_i < s_j$ and $C_i > s_j$



Size represents length
Numbers represent weights

Q. What is the runtime of a trivial algorithm?
A. Try each sequence, so O(n!)

Scheduling with precedence constraints

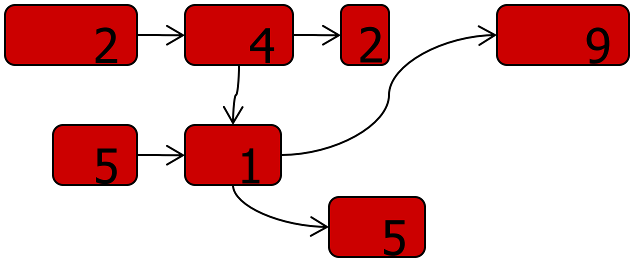
Given

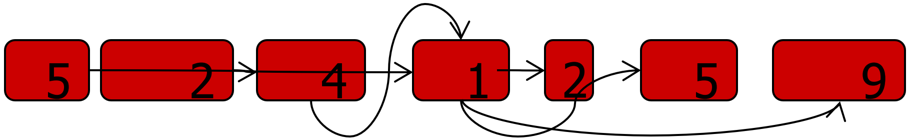
- 1-machine, set J of n jobs, each with a length p_j and a weight w_j
- precedence constraints (partial order), i.e. i precedes j iff $i \rightarrow j$

Find

- non-preemptive schedule with completion times C_j for each job j
- obeying precedence constraints, and with
- minimum sum of weighted completion times $\Sigma_j^n\, w_j C_j$

Schedule:
- start time s_j for every job j
- $C_j = s_j + p_j$
- no jobs $i \neq j$ with $s_i < s_j$ and $C_i > s_j$





Size represents length
Numbers represent weights

TU Delft

Scheduling with precedence constraints

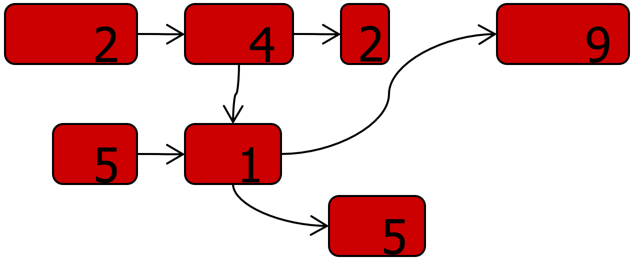
Given

- 1-machine, set J of n jobs, each with a length p_j and a weight w_j
- precedence constraints (partial order), i.e. i precedes j iff $i \rightarrow j$

Find

- non-preemptive schedule with completion times C_j for each job j
- obeying precedence constraints, and with
- minimum sum of weighted completion times $\Sigma_j^n w_j C_j$

Schedule:
- start time s_j for every job j
- $C_j = s_j + p_j$
- no jobs $i \neq j$ with $s_i < s_j$ and $C_i > s_j$



Size represents length
Numbers represent weights

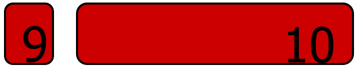
Q. What would be a good heuristic based on the weight?
A. Heavy weight up front, light weights at the end
(when precedences allow)

Example where “lowest weight last” fails

$p_1 = 9$ $p_2 = 1$

penalty (C_1=9, C_2=10):
90 + 90 = 180

$p_2 = 1$ $p_1 = 9$



penalty (C_2=1, C_1=10):
9 + 100 = 109

Scheduling with precedence constraints

Given

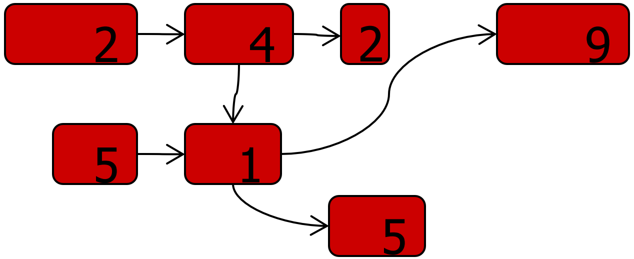
- 1-machine, set J of n jobs, each with a length p_j and a weight w_j
- precedence constraints (partial order), i.e. i precedes j iff $i \rightarrow j$

Find

- non-preemptive schedule with completion times C_j for each job j
- obeying precedence constraints, and with
- minimum sum of completion times $\Sigma_j^n w_j C_j$

Schedule:

- start time s_j for every job j
- $C_j = s_j + p_j$
- no jobs $i \neq j$ with $s_i < s_j$ and $C_i > s_j$



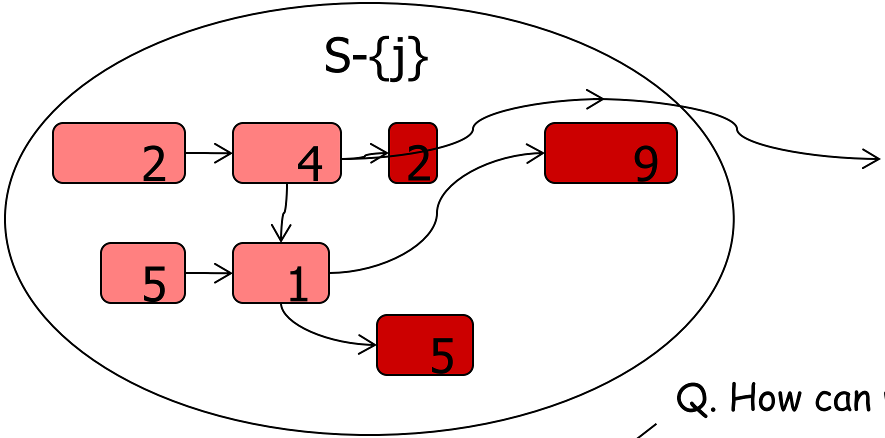
Size represents length
Numbers represent weights

Q*. Where to branch on, or what to consider as a subproblem? (3 min)
A. Branch on which task to be the last one (on which no other depends).
Subproblem: the optimal schedule of the subset of remaining tasks

Scheduling with precedence constraints

Idea. Recurse on each last job j (on which no other jobs depend); take minimum.

Q. What do we need to know to decide on j?



Q. How can we know the completion time of j?

OPT[S] = $\min_{j \in LAST(S)}$ { OPT[S-{j}] + $w_j p(S)$ }
where LAST(S) is set of jobs in S without successor in S and $p(S)=\Sigma_{i \in S} p_i$.

Q. Base? A. OPT[∅]=0

Scheduling with precedence constraints

```
Scheduling(J) {
   p[Ø] = 0
   M[Ø] = 0
   foreach (subset S of J in increasing size) {
         p[S] = p[S-j]+p_j  (for some last job j from S)
         M[S] = min_{j∈LAST(S)} { M[S-{j}] + w_j p[S] }
   }
   return M[J]
}
```

Where **LAST(S)** is the set of jobs in S without successor in S.

Q. What is a tight bound on space and runtime of this algorithm?
A. array is 2^n for all subsets S (also for p[S])
filling it takes time O(n), so $O(n \cdot 2^n) = O^*(2^n)$

NB: *An equivalent solution using FIRST instead of LAST also exists (but mind taking the effect of the length of the first job on the rest into account).*

Woeginger, exercise 33: Scheduling with precedence constraints **and release times**

Given
- 1-machine, set J of n jobs, each with a length p_j *and a release time r_j*
- precedence constraints (partial order), i.e. i precedes j iff $i \rightarrow j$

Find
- non-preemptive schedule with completion times C_j for each job j
- obeying precedence constraints and release times, and with
- minimum sum of completion times $\Sigma_j^n C_j$

TU Delft

Wavelength-Division Multiplexing

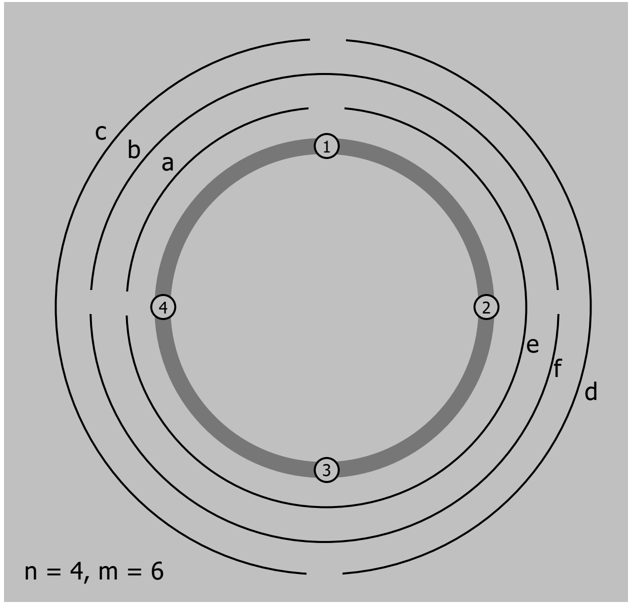
Wavelength-division multiplexing (WDM). Allows m communication streams (arcs) to share a portion of a fiber optic cable, provided they are transmitted using different wavelengths.

Ring topology. Special case is when network is a cycle on n nodes.

Bad news. NP-complete, even on rings.

Brute force. Can determine if k colors suffice in $O(k^m)$ time by trying all k-colorings.

Goal. $O(f(k)) \cdot \text{poly}(m, n)$ on rings.



Wavelength-Division Multiplexing

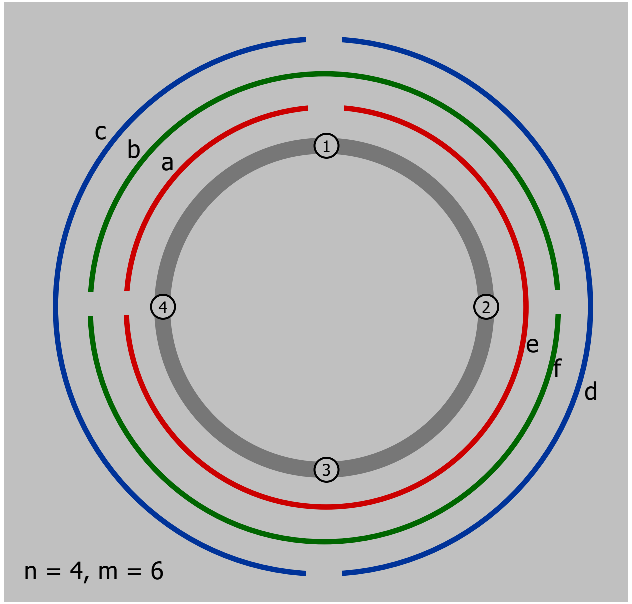
Wavelength-division multiplexing (WDM). Allows m communication streams (arcs) to share a portion of a fiber optic cable, provided they are transmitted using different wavelengths.

Ring topology. Special case is when network is a cycle on n nodes.

Bad news. NP-complete, even on rings.

Q. What is the runtime of a brute force approach?
A. Can determine if k colors suffice in $O(k^m)$ time by trying all k-colorings.

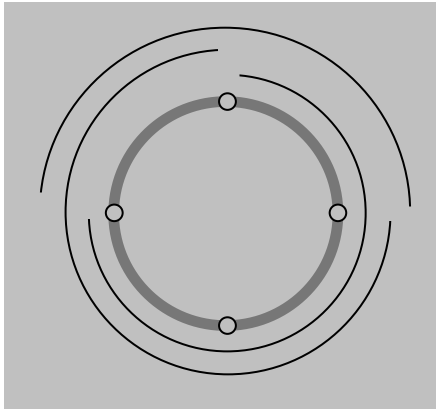
Goal. $O(f(k)) \cdot \text{poly}(m, n)$ on rings.



Circular arc coloring. Given a set of n arcs with depth d ≤ k, can the arcs be colored with k colors?

Q. How many colors do we always need *at least*?

A. at least the number of streams at one location



Q. How many colors do we need for this example?

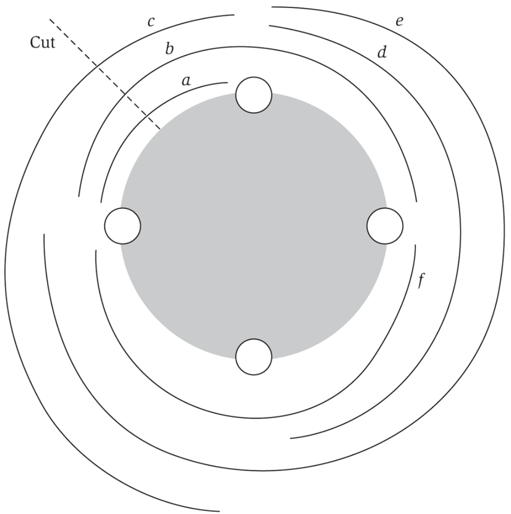
max depth = 2
min colors = 3

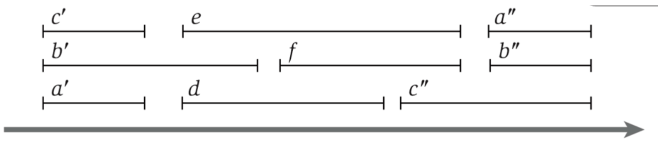
A. three: each pair of the three lines overlaps with the other two

The main idea: re-use a known algorithm

Q. What if this wasn't a graph but just a line?
A. Interval scheduling (coloring), polynomial time algorithm

Q. For a circle, how to use this? What is the problem?



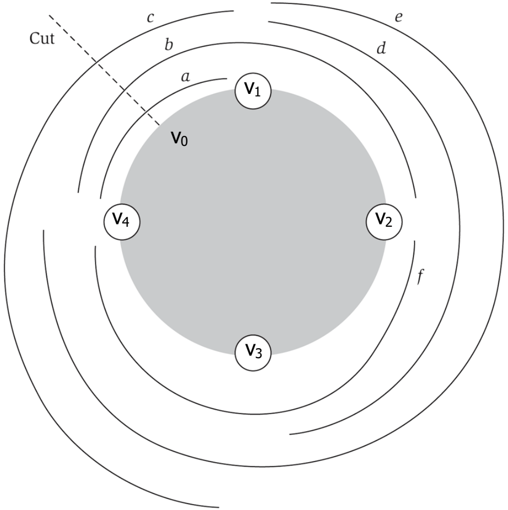


colors of a', b', and c' must correspond
to colors of a", b", and c"

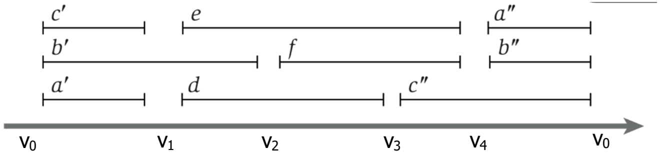
(Almost) Transforming Circular Arc Coloring to Interval Coloring

Circular arc coloring. Given a set of n arcs with depth $d \leq k$, can the arcs be colored with k colors?

Equivalent problem. Cut the network between nodes v_1 and v_n. The arcs can be colored with k colors iff the intervals can be colored with k colors in such a way that "sliced" arcs have the same color.



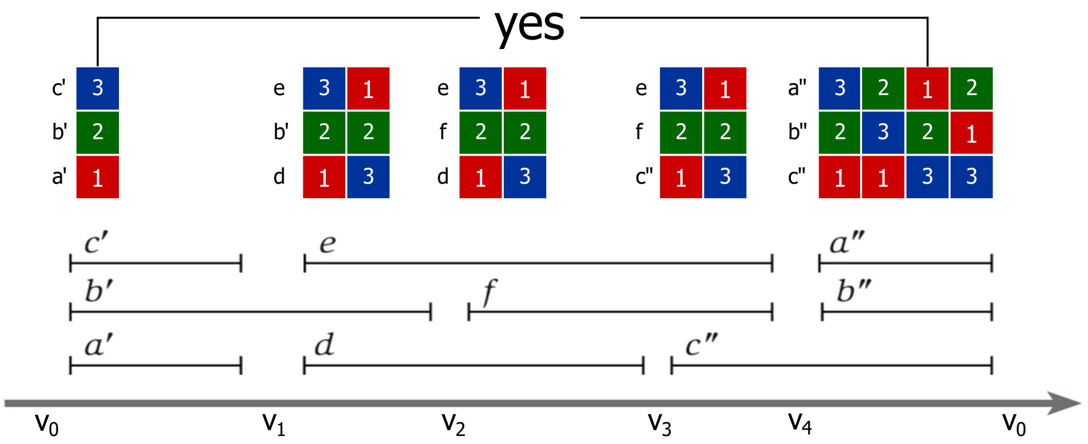
colors of a', b', and c' must correspond
to colors of a", b", and c"



TU Delft

Dynamic programming algorithm.

- F_0 = { assign distinct color to each interval which begins at cut node v_0 }
- Enumerate all k-colorings F_i of the intervals through v_i that are consistent with the colorings F_{i-1} of the intervals through v_{i-1}.
- The arcs are k-colorable iff some coloring of intervals ending at cut node v_0 is consistent with original coloring of the same intervals.



Circular Arc Coloring: Runtime

Q. What is the runtime of this algorithm?
A. $O(k! \cdot n)$.
- n phases of the algorithm.
- Bottleneck in each phase is enumerating all consistent colorings.
- There are at most k intervals through v_i, so there are at most k! colorings to consider.

Remark. This algorithm is practical for small values of k (say k = 10) even if the number of nodes n (or paths) is large.

1-Slide Summary on Dynamic Programming

Traveling Salesperson

$OPT[\{i\};i] = d(1,i)$ for every i

$OPT[S;i] = \min_{j\in S-\{i\}} \{ OPT[S-\{i\};j] + d(j,i) \}$

$\min_{i\in \{2,\ldots,n\}} \{ OPT[\{2,\ldots,n\};i] + d(i,1) \}$

Scheduling with precedences

$OPT[S] = \min_{j\in LAST(S)} \{ OPT[S-\{j\}] + w_j p(S) \}$

where $LAST(S)$ is set of jobs in S without successor in S and $p(S)=\Sigma_{i\in S} p_i$

Circular Arc Coloring

Enumerate all k-colorings F_i of the intervals through v_i that are consistent with the colorings F_{i-1} of the intervals through v_{i-1}.

Is F_n consistent with the coloring in F_0?

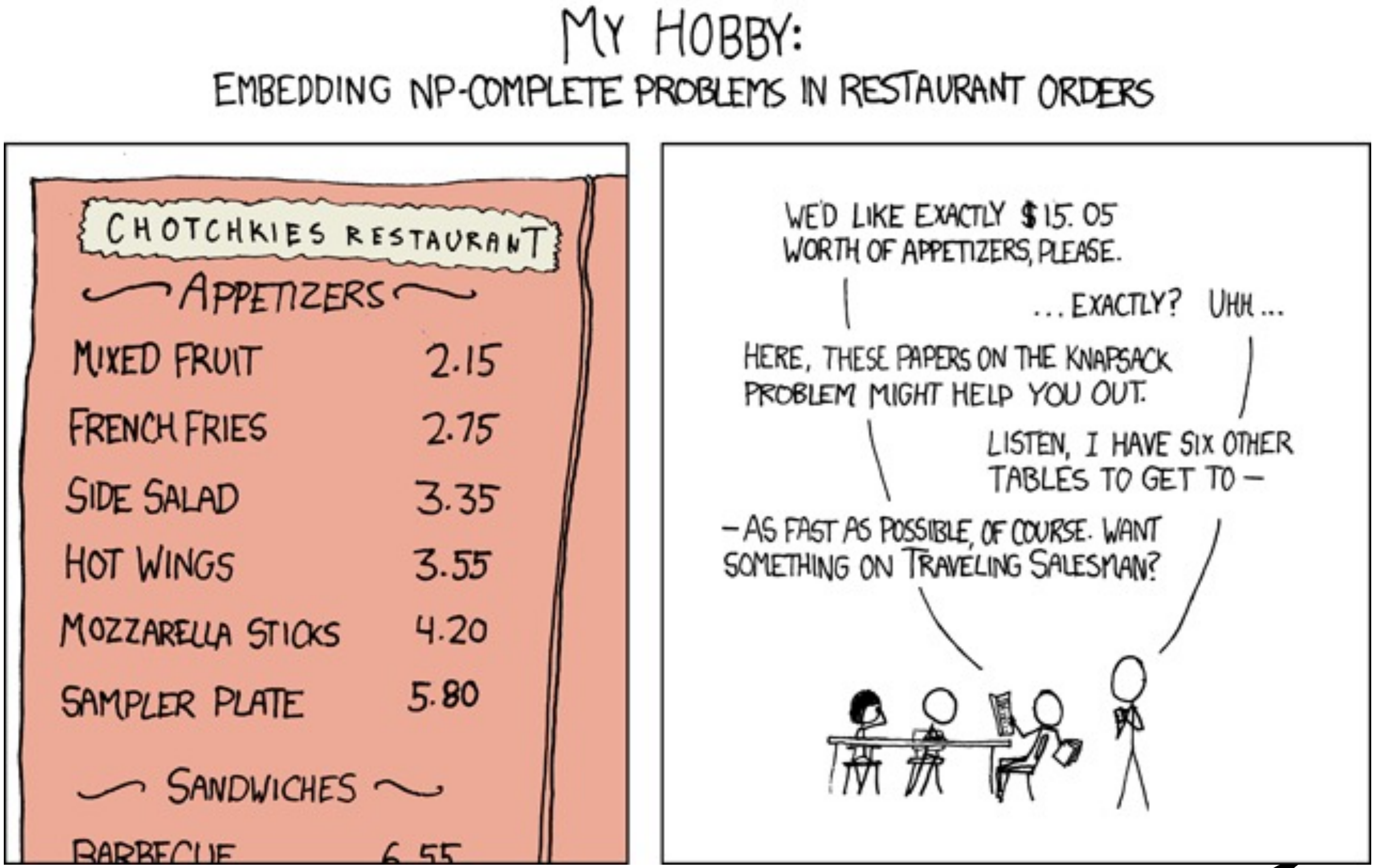
Study Advice

Please read:

1. Section 10.3 from Jon Kleinberg and Eva Tardos, *Algorithm Design*, 2006.
2. Gerhard Woeginger, Exact algorithms for NP-hard problems: A survey, *Combinatorial Optimization*, LNCS 3570, pp 187-207, 2003: Section 4 for DP

Homework assignments

- Dominating set in a graph (BrightSpace)
- Exercise 33 in paper by Woeginger [2]



http://xkcd.com/287/