Dumbshit guide to KO 🦃

Basics

- Polynomial time reduction
 - For any instance of problem A we have a polynomial time algorithm, that can transform it to a problem B
 - If this is possible, than the solution to problem A is the same as the solution to problem B

Integer linear programming

- Branch and bound solver for ILP
- Totally unimodular matrix
 - matrix A is totally unimodular if the determinant of every square submatrix of A is $\in \{-1,0,1\}$
 - ullet if A is unimodular then $a_{ij} \in \{-1,0,1\} \ orall i,j$
 - if a problem is given by a totally unimodular matrix A, then the simplex algorithm will return integer solutions
 - Sufficient condition
 - $ullet \ a_{ij} \in \{-1,0,1\} \ orall i,j$
 - ullet each column contains either one non-zero element or exactly 2 non-zero elements +1 and -1
- Real estate investment & logical conditions
- N-partition
 - 2-partition problem
 - decision problem
 - split loot into 2 piles, which should be equal in value
 - min 0
 - $s.t. \sum_i x_i \cdot p_i = \sum_i (1-x_i) \cdot p_i$
 - where x_i is binary

- minimise difference
 - split loot into 2 piles, which should be as equal as possible
 - $\min C_{max}$
 - s.t.
 - $x_i \cdot p_i \leq C_{max}$
 - $(1-x_i) \cdot p_i \leq C_{max}$
 - where x_i is a binary vector, i.e. $x_i \in \{0,1\} \ orall i$
- Jidelna
- TSP
- Max absolute value
 - we have

$$\min \sum_{i=1}^m |x_i| \ s.\, t.\, Ax \geq b$$

• absolute values are not allowed in ILP or LP so we have to transform it using a proxy variable \boldsymbol{z}

$$egin{aligned} \min \sum_{i=1}^n z_i \ s.\ t. \ Ax &\geq b \ z_i &\geq x_i & orall i \ z_i &\geq -x_i & orall i \end{aligned}$$

Otázky z minulých let

- formulace nejakeho problemu
 - investice do real estate + podminky XOR, OR
 - n-partition
 - vyroba
 - skolni jidelna
 - scheduling, scheduling with temporal constrains
 - TSP s time windows
- Iterace B&B
- Shortest path ILP
 - analogie se spagatky a tenisaky

- max l_t -> cilovy vrchol to tahne co nejvic dolu
- s.t.
 - $l_i \leq l_i + c(i,j)$ -> vrchol muze byt dolu max delku hrany
 - $l_s=0$ -> start vrchol je nahore

Shortest paths

- Path problems
 - SPT (shortest path tree)
 - out-tree
 - path from one vertex to all other
 - in-tree
 - path from all vertices to one
 - shortest path vs. longest path
 - shortest path problem can be transformed to longest path problem by inverting the sign for each edge
 - for the longest path we can not solve it if the graph contains a positive cycle
 - path (cesta) vs. edge progression (sled)
 - path can not repeat edges and vertices, edge progression can, if the graph does not contain negative cycles, edge progression is equal to path
 - triangle inequality for shortest paths: $l(i,j) \leq l(i,k) + l(k,j)$
 - on graphs, where this holds for all vertices, the algorithm can be simpler
 - for example maps
- Bellman's principle of optimality
 - we have a shortest path between s-w with k edges, then the path between s-v (where v and w are neighbours) is the shortest path between s-v with k-1 edges
 - shortest path between two vertices is composed of shortest paths between the vertices on the path

Bellman equation

$$l(s,v) = \min_{k
eq v} \left\{ l(s,k) + c(k,v)
ight\}$$

- Dijkstra
 - Characteristics
 - can not handle negative weights nor negative cycles
 - computes a SPT (shortest part tree) for a starting vertex
 - time complexity
 - min heap $O(E \log V)$
 - because every time we add an edge to the min heap we need to sort it this happens in time

$$ullet \ \log E \leq \log(V \cdot (V-1)) = \log V^2 = 2 \log V = \log V$$

- in the worst case we will do this for each edge so
 - $O(E \cdot 2 \log V) = O(E \log V)$
- Fibonacci heap $O(E + V \log V)$
- Algorithm

```
# --- Near python PseudoCode ---
queue = MinQueue() # based on cost
visited = [False * num vertices]
predecesors = [NaN * num vertices]
costs = [∞ * num_vertices]
queue.push(starting_vertex)
while not queue.empty():
        # out of unvisited vertices
        # extract the one with the
        # closest distance to start
        v = queue.pop()
        visited[v] = True
        for each neighbour n of v:
                # if already visited
                # it is no longer possible
                # to find a shorter path to this
                if visited[n]:
```

Proof of optimality

- We have a set of visited vertices R, which contains shortest distances
- Initially we add to the set R the start vertex, for which the distance of 0 is trivially optimal, now assuming that R contains optimal distances (which it does after this 0-th step)
- We have the set of visited vertices R for which we know we have the shortest path, because for predecessors $k \in R$ we can apply Bellman's equation
- The core of the proof is that when we expand R by a neighbouring vertex $n \notin R$ with minimal distance from vertices $v \in R$ we are applying the Bellman equation in the form of d(n) = d(v) + l(v,n) and thus we are getting the shortest path to n
- The value for predecessors of n not in R ($k \notin R$) can decrease, however because we have non-negative edge weights it can not get smaller, than the value for the predecessors $k \in R$
- In conclusion once we add a vertex to R we can no longer find a shorter path to it

Bidirectional Dijkstra

- we have start and end node
- from the start node we run a forward search
- from the end node we run a backward search
- stop when the two sets intersect

Bellman-Ford

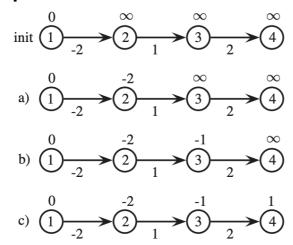
Characteristics

- can handle negative weights but not negative cycles
- time $O(V \cdot E)$ is given by the two loops, which is worse than Dijkstra

- for DAG we can use topological ordering and compute the shortest path quickly using dynamic programming
- bellman ford can also be modified to detect negative cycles

Algorithm

Example



Proof of optimality / correctness

• for bellman ford we need to prove that in iteration k the label l assigned by the algorithm to vertices which are k edges far from the initial vertex is optimal

• theorem about k^{th} iteration of bellman ford

• in the k^{th} iteration of the outer loop the label $l^k(w)$ assigned to vertex w is better or worse than the shortest path from s to w $P^k(s,w)$ which uses k vertices with length $c(E(P^k(s,w)))$

$$l^k(w) \leq c(E(P^k(s,w)))$$

- for example with lucky edge ordering the algorithm might finish in the first iteration
- if this theorem holds it would mean that in k=V(G) iterations we will get a correct shortest path tree for the entire graph G
- now we need to prove the theorem

- for k=1 it holds trivially that $l^1(w) \leq c(s,w)$
- so using induction hypothesis, in iteration k-1 we have $l^{k-1}(w) \leq c(E(P^{k-1}(s,w)))$
- then in k^{th} iteration, when we are examining edge (v,w) in the internal loop it holds that $l^k(w) \leq l^{k-1}(v) + c(v,w)$
- Then using the theorem that P^k is the shortest path and bellman principle of optimality it holds that $c(E(P^{k-1}(s,v)))+c(v,w)=c(E(P^k))$
- putting it all together

$$l^k(w) \leq l^{k-1}(v) + c(v,w) \leq c(E(P^{k-1}(s,v))) + c(v,w) = c(E(P^k))$$

• and because no path is shorter than the shortest one and no path can have more than n-1 edges, at iteration k=n-1 we can compress the above formula into

$$l^k(w) = c(E(P^k))$$

Floyd

Characteristics

- can handle negative weights
- produces a matrix of shortest paths between all verticies
- and also a matrix of predecessors for each vertex, we can use this to reconstruct the paths
- start with adjacency matrix, weight where there is an edge, ∞ where there isn't one
- a loop for each vertex k
- look whether we can use this vertex to shorten the path between all vertices i,j by taking $i \to k \to j$ instead

Algorithm

```
cost(i,j) = cost(i,k) +
cost(k,j)

predecessor(i,j) =

precessor(k,j)
```

Otázky z minulých let

- Iterace Djikstry
- Dukaz spravnosti pro Djikstru
- Formulace problemu aby sel Djikstrou

Max reliability network

- · communication network with prob of failure
- the max reliability path is given by the highest product of probabilities, however regular Dijkstra considers the sum of edge weights, not the product
- so we can either modify the weights by taking their $-\log(w)$, (though this will only work for weights $w \in [0,1] \setminus 0$, however this is not a problem for probabilities)
- or we can modify the algorithm by using max queue, swap sum with product and < with > and initialize properly, if we re-did the proof for this we would find that we cannot have w>1, for the same reason we cannot have negative weights in normal Dijkstra

Prelevani vody s 3 a 5 l flaskami, dostat 4 l

- represent all states, where at least one bottle is either full or empty 0/0,...,3/0,...,0/5,...,3/5
- draw edges between neighbouring states
 - edge weights can be one if we want to minimize the number of manipulations
 - or the liters of wasted water if we want to minimize wasted water
 - or the manipulated water if we want to minimize the liters of manipulated water
- run Dijkstra and compute the distance to the vertices, which give together 4, so either 3/1 or 0/4

Mosty a pochodne

 most s dirama a zraneny vojaci, potrebuji baterku aby se dostali pres most ale maj jen jednu, takze vzdycky prejdou 2 a jeden se musi vratit

- prejdou most za 1,2,5,9 minut, ale kdyz jdou dva musi se pohybovat tempem toho pomalejsiho
- zvladnou prejit za <16 minut?
- draw all combinations and their edges
- solve with Dijkstra

Trucks

Investment

- we have 15 years and want to invest money and make as much as possible
- each opportunity has a starting year a maturity period and an appreciation (expressed as percentage)
- we want to invest everything into the one opportunity which maximizes the profit
- we can collect and reinvest the money after it matures

Solution

- 15 vertices for 15 years
- we want to model compound interest, so we take the $w=-\ln(1+\frac{a}{100})$ (negative because we want the longest path) of the appreciation interest and use it as weights instead
- for each opportunity draw an edge (e.g. it starts at 1 and matures at 6, then the edge will be (1,6) with corresponding weight)
- draw an edge with zero appreciation rating between each year (to model if we do not invest in a year)
- Fire stations
- Warehouses

Flows

Task formulation

- vstup (G, l, u, s, t)
- G vstupni graf

- $ullet l:E o R_0^+$
 - lower bound na toky pro jednotlive hrany
- $ullet u:E o R_0^+$
 - upper bound na toky pro jednotlive hrany
- s startujici vrchol
 - vetsinou nema omezeni na to, co z nej vytece
- t cilovy vrchol
 - vetsinou nema omezeni na to, co do nej vtece
- pro vsechny uzly krome zdroje a odtoku plati kirchoffovy zakony

Max flow problem

- Can also be formulated as a LP, but this is less efficient, than using a specified algorithm, as LP is a general method and we do not put any information about the problem into the model
- max the flow coming out of start minus the flow coming into start
- $ullet \max \sum_{e \in \delta^+(s)} f(e) \sum_{e \in \delta^-(s)} f(e)$
- s.t.
 - for each vertex except source and sink the Kirchhoff law must hold

$$ullet \ \sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e) = 0 \quad orall v \in E(G) \setminus \{s,t\}$$

- for each edge the lower-bounds and upper bounds must be satisfied
 - $ullet \ l(e) \leq f(e) \leq u(e) \quad orall e \in E(G)$

Ford-Fulkerson algorithm

- Start with some initial flow, for a graph with zero lower bounds flow of 0 is also a valid flow
- Find path from start to sink using DFS
 - we can walk edges both in their positive direction (if there is flow to add)
 - and in their negative direction (if there is flow to subtract)
- Find the bottleneck capacity of this path
 - Add the bottleneck capacity to positive edges on the path
 - Subtract the bottleneck capacity from negative edges on the path
- The bad thing about FF is that the time complexity is dependent on the upper bounds in the network, e.g. if it consistently finds low increments for large upper bounds it can take long
- Initial feasible flow for FF for graph with non-zero lower bounds

- FF needs a initial flow to improve on
- In graph with 0 lower bounds the initial flow can be 0, however in graph with non-zero lower bounds this is not so simple

Non-zero lb -> zero lb transform

- graphs G, G', init G' := G
- add two vertices s',t' to G', add an edge t o s in G' with $lb=0,ub=\infty$
- take the old graph and modify each edge
 - $egin{aligned} \bullet & orall e \in E \ & \bullet & lb'(e) := 0 \ & \bullet & ub'(e) := ub(e) lb(e) \end{aligned}$
- for each vertex v in G, compute

$$ullet L(v) = \sum_{(i,v) \in E} lb(i,v) - \sum_{(v,j) \in E} lb(v,j)$$

- add edges to G'
 - (s',v), where L(v)>0• lb(e)=0, ub(e)=L(v)
 - $ullet (v,t'), ext{ where } L(v) < 0 \ ullet (b(e) = 0, ub(e) = -L(v)$

• Find max flow in G'

- The initial flow can now be all zeros
- Run Ford-Fulkerson in the G' graph
- If the max-flow in the G' graph saturates all edges (s',i), meaning that f(s',i)=ub(s',i)=L(v)
- Take the resulting flow f' in G' and transfer it into initial flow f in G like

•
$$f(e) := f'(e) + lb(e)$$

If such flow exists this is an initial flow in G and can be used to run
 FF on it, if it does not exist, graph G does not have a feasible flow

Edmonds karp algorithm

- We can improve Ford-Fulkerson by using BFS instead of DFS when searching for paths this gives us always the shortest path in terms of edges
- $O(|E|^2 \cdot |V|)$

Minimum capacity cut (min cut)

- after Ford-Fulkerson finishes
- run one last bfs, draw a dumpling around the nodes we reached
- edges which cross the dumplings borders form the minimum capacity cut
- there can be more than one minimum capacity cut
- it forms the bottleneck of a network

Theorem

- the value of the maximum flow between source and sink is equal to the capacity of the minimum cut
- i.e. the max flow through the network has to be equal to the bottleneck in the network, which doesn't sound too revolutionary

Integrality

 if the capacities of the network are integers and there exists a feasible flow, then the maximum flow is an integer as well

Feasible flow with balances

 multiple sources and sinks, we want to decide whether the production from source nodes is able to satisfy the demand in the sink nodes

Decision problem

- we have a problem (G, l, u, b)
- where $b(v) \ \forall v \in V(G)$, represent the supply / consumption of each node, i.e. a balance
- ullet and $\operatorname{sums}-\operatorname{consumptions}=0$ or equivalently $\sum_{v\in V(G)}b(v)=0$
- than we want to decide whether there exist a flow such that

$$\sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e) = b(v); \quad orall v \in G(V)$$

Reduction to maximum flow

- reduce to max flow by adding
 - super-source, connect it to sources with edges whose lower bound is the balance of that vertex
 - super-sink, connect the sink nodes to it with lower bounds given by the demand and see wether we can run max flow on it

 connect them to the existing sources and sinks with edges, whose up are the original balances

Minimum cost flow

- add cost to each edge, e.g. we have to pay for each unit that flows through each node
- we still want max flow but with minimum cost
- Min cost flow LP formulation

$$egin{aligned} \min \sum_{e \in E(G)} c(e) \cdot f(e) \ &s.t. \sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e) = b(v) \qquad v \in V(G) \ &l(e) \leq f(e) \leq u(e) \qquad \qquad e \in E(G) \end{aligned}$$

Max flow → min cost flow

- add edge (t,s) with $u=-\infty, c=-1$
- set cost for all other edges to 0 $\forall e \setminus (t,s), c = 0$
- this is minimum cost circulation maximizes the flow on the added edge

• Shortest path o Min cost flow

- use LP for min cost flow
- b(s) = 1, b(t) = -1
- for all other edges b(v) = 0
- for all edges l(e) = 0, $u(e) = \infty$

Min cost flow cycle canceling algorithm

- input : network (G, l, u, b)
- 1 : find feasible flow by solving feasible flow with balances
- ullet 2 : build residual graph G_f wrt f
 - $\bullet \ \ G_f = (V, E_f, u_f, c_f)$
 - same vertices, modified set of edges, modified upper bounds and modified costs, no lower bounds
 - · add backward edges to the graph
 - update upper bound
 - forward edges $u_f(i,j) = u(i,j) f(i,j)$
 - backward edges $u_f(j,i) = f(i,j) l(i,j)$
 - remove edges, whose upper bounds end up 0

- update costs
 - forward edges $c_f(i,j) = c(i,j)$
 - backward edges $c_f(j,i) = -c(i,j)$
- 3 : find a negative cost cycle in G_f , if none exists, stop

Finding a negative cost cycle

- residual graph (G_f, u_f, c_f) , where all edges have $u_f(e) > 0$
- add a source node s and connect it to every node in the residual graph with edge of zero cost
- use e.g. Bellman-Ford to detect a negative cost cycle wrt to cost c_f
- return negative cycle or report that it does not exist
- $4: \mathsf{compute} \; \gamma = \min_{e \in E(C)} u_f(e)$
 - e.g. the minimal upper bound across the negative cycle
- 5: Augment f along C by γ
 - increase along forward edges, decrease along backward edges
 - the criterion will decrease by γ · value of the negative cost cycle
- 6: Goto 2

Time complexity

- U max upper bound in G, C max cost in G
- The flow will then be $-|E|\cdot C\cdot U \leq c(f) \leq |E|\cdot C\cdot U$
- Each iteration improves the flow by at least 1
- So then we will have at most $2\cdot |E|\cdot C\cdot U$ iterations and in each iteration we run Bellman-Ford to find the negative cycle ($O(|V|\cdot |E|))$
- Which gives us $O(|V| \cdot |E|^2 \cdot C \cdot U)$

Improvement

• by using an algorithm which finds the mean negative cost cycle we can get the complexity $O(|V|^2 \cdot \log |V| \cdot |E|^3)$

Minimum cost multi-commodity flow

- we have multiple sources and sinks per commodity
- commodities can't mix, so for each vertex Kirchhoff law has to hold for each commodity independently

Instance

• $(G, l, u, c, b^m, \dots, b^M)$

- where b^m is a balance vector, that specifies the balance for each commodity
- we have a separate balance for each commodity in each vertex
- find a feasible flow, such that the cost is minimal
- LP formulation

$$egin{aligned} \min \sum_{e \in E(G))} \sum_{m \in M} f^m(e) \cdot c(e) \ s.\,t. \ &\sum_{e \in \delta^+(v)} f^m(e) - \sum_{e \in \delta^-(v)} f^m(e) = b^m(v) \qquad v \in V(G), m \in M \ &l(e) \leq \sum_{m \in M} f^m(e) \leq u(e) \qquad \qquad e \in E(G) \end{aligned}$$

Matching

- Maximum cardinality matching
 - as many matches as possible
- Maximum cardinality matching in bipartite graphs
- Minimum weight matching
- Minimum weight perfect matching

Otázky z minulých let

- Iterace FF + min cut
- Flow scheduling $P2|r,d,preempt|C_{min}$

3 parallel identical resources

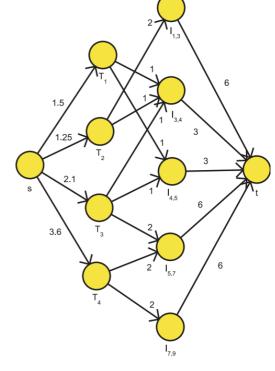
\dot{T}_j	1	2	3	4
p_j	1.5	1.25	2.1	3.6
r_j \widetilde{d}_i	3	1	3	5
\widetilde{d}_i	5	4	7	9

1) Nodes $I_{1,3}$, $I_{3,4}$, $I_{4,5}$, $I_{5,7}$ a $I_{7,9}$ represent time interval in which the tasks can be executed (intervals are given by r and d).

E.g. $I_{1,3}$ represents time interval (1,3).

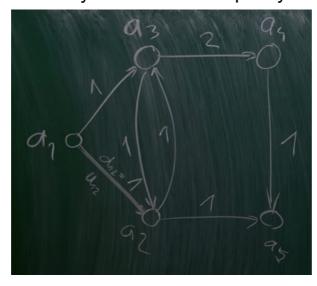
- 2) The upper bound for the $(T_i, I_{x,y})$ arc is the length of the time interval (i.e. y - x), since the tasks are internally sequential.
- 3) The upper bound for $(I_{x,y}, t)$ is the length of the interval multiplied by the number of





Dynamic flow

- task
 - a_1, \ldots, a_n cities
 - with q_1, \ldots, q_n cars
 - should be transported to city a_n in K hours
 - each edge has capacity u_{ij} and length d_{ij}
 - each city has a limited capacity for cars p_i



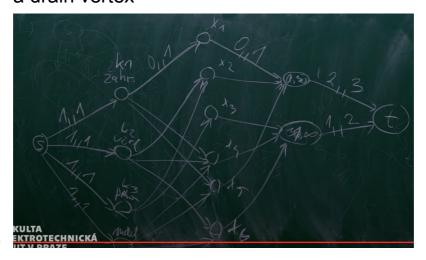
solution

- we have a graph with the cities and roads
- we model the time by
 - having a x axis with time and y axis with graph vertices for each time tick $a_{(1,\ldots,n),(1,\ldots,k)}$
 - add edges i o i with parking capacity p_i

- add time edges between cities, where it goes across as many columns as is the time length of the edge
- do this up to K
- add a starting vertex which connects to all cities in the first layer with capacities q_i (can be lower bound if we want to answer can we get them all to a_k)
- max flow between start s and drain a_{nk}

Distinct representatives

- n residents in a city and each of them belong both to at leat one club k_1,\ldots,k_l and age group p_1,\ldots,p_r
- each club nominates a member to the town council
 - the council has a limit on how many members of each age group can be on it
 - a resident can represent only one club
- solution
 - start vertex with capacities lb=1, ub=1 that lead to clubs (each club has one representative)
 - a column with vertices for clubs connected to residents
 - a column with vertices for residents connected to age groups
 - a column with vertices for age groups connected to drain vertex, where again we have some lb and ub
 - a drain vertex



Dining problem

- task
 - p families
 - a(i) members from $i=1,\ldots,p$
 - ullet t tables with limited capacity $b(j)\ orall j \in \{1,\ldots,t\}$

- solution
- Rounding in 3x3 matrix
 - start
 - 3 vertices for row sums
 - 3 vertices for col sums
 - drain
 - $s \xrightarrow{ ext{row sums floor,ceil}} ext{row sums} \xrightarrow{ ext{matrix entry floor, ceil}} ext{col sums} \xrightarrow{ ext{col sums floor,ceil}}$

Transport problem

- we have cities with trailers and a truck
- if we drive from a city A that has a trailer for city B to city B we get some
 - loss = cost reward
 - (negative edge, reward should be larger than cost)
- if we drive from A to C and A doesn't have a trailer for C there is
 - loss = cost 0(zero reward)
- we want to find a path trough the graph, that minimizes the loss (e.g. max profit)

Knapsack

- A pack with limited capacity, the sum of weight of all items is bigger than the capacity, fit as many items in the pack so that the cost of the items is maximal
- Formally
 - Weight limit W
 - Set of n items with cost and weight $(c, w) \in I$
 - Goal:
 - find a subset $S \subseteq \{1,\ldots,n\}$ such that $\sum_{j \in S} w_j \leq W$
 - $\max \sum_{i \in S} c_i$

Fractional knapsack

- allow the division of items and solve a modified knapsack problem
- this problem is polynomial
- Algorithm
 - order by relative cost $\frac{c_i}{w_i}$

- stuff them in the backpack and find the first one which doesn't fit
- cut of the part which doesn't fit
- sorting is $O(n \cdot \log n)$, scanning is $O(n) \to \mathsf{time}$ is $O(n \log n)$
- could be formulated by a ILP

Approximation algorithm

- for all instances I of the problem
- maximalization problem
 - assurance, that our algorithm will not be worse, than some fraction of optimal solution
 - $ullet J^A(I) \geq rac{1}{r}J^*(I)$
- minimalization problem
 - $ullet r \cdot J^*(I) \geq J^A(I)$
 - assurance, that our algorithm will not be worse, than some multiple of optimal solution

2-approximation algorithm for Knapsack

- find the solution for fractional knapsack
- out of the solutions which contain either $(1,\ldots,h-1)$ or (h) select the better solution

Proof for approximation factor 2

- we can omit items with w > W
- if $\sum_i w_i \leq W$ then the solution is trivial
- From fractional knapsack we can tell that $\sum_{i=1}^h c_i$ forms an upper bound on the solution (optimal solution will be lower than this by the amount the h^{th} item can't fit into the Knapsack)
- Since we select the better cost out of $\left\{\sum_{i=1}^{h-1}c_i,c_h\right\}$, we will get at least half of the optimum bound

$$ullet J^A(I) \geq \max\left\{\sum_{i=1}^{h-1} c_i, c_h
ight\} \geq rac{\sum_{i=1}^h c_i}{2} > rac{1}{2}J^*(I)$$

Dynamic programming for integer weights 0/1 knapsack

- N rows, one for each item
- ullet C_w columns for combinations of weights we can reach up to W

Description

- In each step:
 - Look one field up if we do not want to include this item

- ullet Look one field up and w fields to the left if we want to add this item
- Select the maximum out of these two
- Once the table is filled select the maximum in the last row
- And backtrack
 - in each row if the value in this column is the same as in the column one row above, we do not include this item
 - if they differ, we include it and jump to the left by the weight of this item

Algorithm

- init first row to $-\infty$ except the first column with zero
- For each row $i \in (1, \dots, N)$ (for each item)
 - For each column $j \in (1, \ldots, C_w)$ in the row i

$$ullet x[i,j] = \max(x[i-1,j],\ x[i-1,j-w] + c[i])$$

- Optimal cost
 - $\max x[N,:]$
 - for each row in $i \in (N,\ldots,1)$
 - if x[i,j] == x[i-1,j]
 - do not include this item o continue with x[i-1,j]
 - else
 - ullet include this item o continue with x[i-1,j-w]
- Time $O(n \cdot C_w)$, where C_w are all the possible combinations of weights, this makes the algorithm pseudo-polynomial

Dynamic programming for integer costs

- init first row to ∞ except the first column with zero
- in each row we are adding one new item, explore the combinations of costs reachable via adding the new item and keep the one with the lowest weight for each cell

Approximation scheme for knapsack

- we can speed up the algorithm by running the 2-approximation algorithm and using it as an upper bound on the cost
- compute $t=\max\left\{1,\frac{\epsilon c(S_1)}{n}\right\}$, where ϵ is the tolerance how far of can the dynamic programming be from the true result
- compute new costs $c_j' = \operatorname{round}\left(\frac{c_j}{t}\right)$

- ullet run the dynamic programming with upper bound of $C_{max}=\epsilonrac{2\cdot c(S_1)}{t}$
- Otazky z minulych let
 - vyresit pomoci dyn. prog.
 - 2-aproximacni algoritmus, pseudokod + aproximacni faktor

Travelling salesman problem

- Existence of a hamiltonian cycle
 - Undirected graph G
 - Goal:
 - decide if a Hamiltonian circuit exists
 - NP-complete
- Traveling salesman problem
 - Complete undirected graph with positive weights G
 - directed graph → asymmetric TSP
 - Goal:
 - decide if a Hamiltonian circuit exists
 - NP-complete
- Strongly NP-hard problems
 - Polynomial time
 - Time complexity is polynomial wrt to the size of the input (number of bits needed to represent it)
 - Pseudo-polynomial time
 - If the time complexity is polynomial wrt. to the largest numeric value in the input (largest integer at input) but not necessarily wrt to the size of the input
 - Weakly NP-hard
 - There exists a pseudo-polynomial algorithm for solving the problem
 - Strongly NP-hard
 - There doesn't exist a pseudo-polynomial time algorithm
 - Optimization problem L
 - For a polynomial p let L_p be the restriction of L to such instances I that consist of nonnegative integers with $largest(I) \leq p(size(I))$, i.e.

- numerical parameters of L_p are bounded by a polynomial in the size of the input.
- L is called strongly NP-hard if there is a polynomial p such that L_p is NP-hard. If L is strongly NP-hard, then L cannot be solved by a pseudo-polynomial time algorithm unless P = NP.
- i.e. a problem is NP-hard if an smaller instance with inputs limited by a polynomial function of the instance size is still NP-hard, if this property exists we can use some dynamic programming algorithm to solve the polynomial subproblems and build a solution

Metric TSP

- TSP with triangle inequality, more formally
- $ullet c: E(G)
 ightarrow \mathbb{R}^+$ such that $c(\{i,j\}) + c(\{j,j\} \geq c(\{i,k\})) \; orall i, j,k \in V(G)$
- still NP-hard, however approximation algorithms do exist

R-approximation algorithm

• algorithm which is able to solve a task with a guarantee that the found solution will be at worst r times worse than the optimum

Proof of TSP being strongly NP-hard

- We have a graph G in which we want to find Hamiltonian cycle (which is a NP-hard task)
- Transform G into a TSP instance G' (TSP instance has a complete graph, so add edges which would make it a complete graph)
 - edges that were in G get weight 1
 - new edges have weight 2
- If an algorithm was able to solve the TSP instance optimally it would mean that it it would have to use the edges with weight 1 only, which in turn would mean that it would be able to solve the task of finding Hamiltonian cycle in a graph, which is NP-hard
- So even if we limit the size of the input to only 2 numbers the problem is still NP-hard and thus TSP is strongly NP-hard

Likely nonexistence of polynomial r-approximation algorithm for general TPS

- proof by contradiction
- We have a graph G in which we want to find Hamiltonian cycle (which is a NP-hard task)

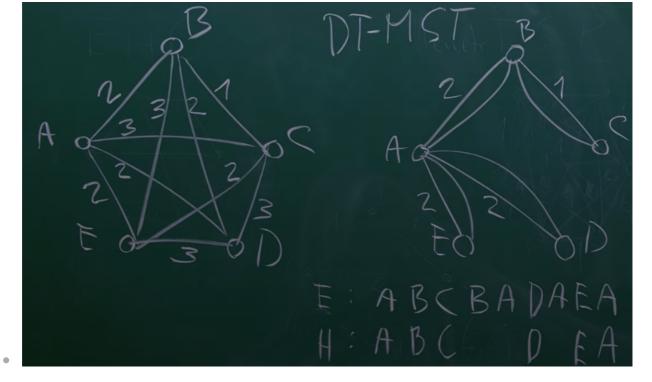
- Transform G into a TSP instance G' (TSP instance has a complete graph, so add edges which would make it a complete graph)
 - edges that were in G get weight 1
 - new edges have weight M, where 1 << M
- if an algorithm solves this the result can be in
 - ullet $[n,M+(n-1)-\epsilon]$: if the solution contains only edges with weight 1
 - ullet >M+(n-1) : if it took at least one of the added edges with weight M
- ullet we make M polynomial in the approximation factor r
 - $\bullet \ \ M=2+(r-1)\cdot n$
 - now if the result is in $[n, n \cdot r]$ there is a Hamiltonian cycle in G
 - and if it is in $[r \cdot n + 1, \infty]$ there is not Hamiltonian cycle in G
- So the conclusion is that an approximation algorithm for any r would need to be able to solve the problem of Hamiltonian cycle, which is NPhard

Heuristic, nearest neighbour

- in each step choose an unvisited neighbouring vertex with minimal cost
- start in vertex vn
- for i in range(2,n):
 - for j in unvisited:
 - choose v[j], such that dist(vn, v[j]) is minimal
 - mark v[j] as visited
 - vn=v[j]
- Complexity $O(n^2)$

Double tree algorithm (DT algorithm)

- Instance (K_n, c) of metric TSP
- Algorithm
 - find a MST T in K_n
 - ullet double every edge in T
 - find a Eulerian walk in T
 - ullet Transform eulerian walk on T into hamiltonian cycle by deleting duplicates from the walk



• Time $O(n^2)$

Approximation factor

- 1. skipped nodes do not prolong the route (triangle inequality) $c(E(L)) \geq c(E(H))$
- 2. deleting edge in the circuit creates a tree so $OPT(K_n, c) \geq c(E(T))$
- 3. doubling the edges $2 \cdot c(E(T)) = c(E(L))$
- putting all this together

$$2 \cdot OPT(K_n, c) \ge 2 \cdot (E(T)) = c(E(L)) \ge c(E(H))$$

Christophides algorithm

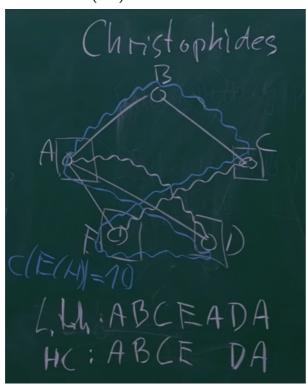
Metric TSP, MST,

• Instance (K_n, c) of metric TSP

Algorithm

- Find a minimum weight spanning tree T in K_n
- ullet W is a set of all odd degree vertices in T
- Find a minimum weight matching M of nodes from W in K_n
 - · Hungarian algorithm is used for this
- Merge T and M to create a multigraph $(V(K_n), E(T) \cup M)$
- Find a Eulerian walk in the merged graph
- Transform the walk into a Hamiltonian cycle (same as in the DT algorithm)

• Time is $O(n^3)$



Approximation factor

- 1. skipped nodes do not prolong the route (triangle inequality) $c(E(L)) \geq c(E(H))$
- 2. deleting edge in the circuit creates a tree so $OPT(K_n, c) \geq c(E(T))$
- 3. The perfect matching considers every 2^{nd} edge, and because it is minimum it chooses the smaller half so $\frac{OPT(K_n,c)}{2} \geq c(E(M))$
- putting all this together

$$\frac{3}{2} \cdot OPT(K_n,c) \geq c(E(T)) + c(E(M)) = c(E(L)) \geq c(E(H))$$

Local search

 improve the initial solution by trying small adjustments and seeing whether they improve the solution

k-OPT local search for TSP

- remove a k-set from the Hamiltonian cycle
- try all the possible combinations to complete the cycle
- if the objective function improved keep the new cycle
- for example for k=2 there is only one combination

Otazky z minulych let

- Chirstophides, algorithmus, casova slozitost, aproximacni faktor
- Pravdepodobna neexistence r-aproximacniho algoritmu pro obecne TSP
- HC reduction to TSP, prove that TSP is np-hard

Scheduling

Task definition & terminology

- set of tasks $au = \{T_1, \ldots, T_n\}$
- optimality criterion γ
 - · max schedule length
 - · sum / weighted sum of completion times
 - minimize lateness

set of parameters

- capacities c_j
- release times r_j
- processing times p_j
- due dates d_j : should be finished
- deadlines \widetilde{d}_j : must be finished

• preemptive tasks:

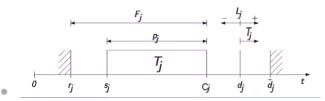
task can't be paused once it started

precedence:

- tasks have to be completed in certain order
- in-tree, out-tree, chain, temporal constrains

variables

- start time s_j
- completion time C_j
- flow time $F_j = C_j r_j$
- lateness $L_j = C_j d_j$
- tardiness $T_j = \max\{C_j d_j, 0\}$



Gannt diagram

sequence of boxes which describe the order of tasks

Graham notation

resources | tasks | criterion

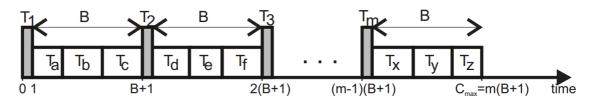
- $P2|pmtn|C_{max}$
 - two parallel processors, preemtion allowed, optimize the completion time for all tasks
 - α -resources
 - parallel resources
 - task can run on any resource
 - dedicated resources
 - specific tasks require specific resource
 - project scheduling
 - m resource types with capacity R_k i.e.

Scheduling on one resource

- Proof of NP-hardness for $1|r_j, \widetilde{d}_j|C_{max}$
 - 3-partition problem
 - strongly NP-hard
 - proof by polynomial reduction from 3-partition problem, which is also strongly NP-hard
 - we have m bins of equal size B
 - each bin contains three items a_i , whose size is $\frac{B}{3} < a_i < \frac{B}{4}$, and all bins are full and contain 3 items $\sum_{i=1}^{3m} a_i = mB$
 - ullet partition the items into m sets such that the size of each set is B

Reduction

- formulate the bin edges as tasks with length 1
- now if the algorithm is able to place the items such that the last item ends at $C_{max}=m(B+1)$ the three partition is possible, if it is larger than the problem is infeasible, in this formulation we have scheduling with release and end dates formulated as 3-partition problem and since 3-partition is NP-hard, scheduling with releases and deadlines is also NP-hard



• Position based ILP formulation for $1|r_j,\widetilde{d}_j|C_{max}$

ILP can solve it optimally, though only for a limited instance size

- this formulation is able to process larger instances than relative order formulation
- variables
 - x_{iq} $(n \times n)$: binary matrix tasks by task order, where each row and column has one 1 and the column in which the 1 is specifies the order of the task
 - C_{max} : end time
 - $t_{q}\left(n
 ight)$: integer / real vector of start times
- constraints:
 - one task at a time

$$\bullet \quad \sum_{q=1}^n x_{iq} = 1$$

$$\bullet \quad \sum_{i=1}^n x_{iq} = 1$$

start time bigger than release data

$$ullet t_q \geq \sum_{i=1}^n r_i \cdot x_{iq}$$

no overlap between tasks

$$ullet t_q \geq t_{q-1} + \sum_{i=1}^n p_i \cdot x_{iq}$$

deadline

•
$$t_q + \sum_{i=1}^n p_i \cdot x_{iq} \leq \sum_{i=1}^n d_i \cdot x_{iq}$$

- C_{max} is the time after last task ends
 - i.e. the start time of the last task + its processing time

$$ullet C_{max} \geq t_n + \sum_{i=1}^n p_i \cdot x_{iq}$$

- criterion:
 - ullet $\min C_{max}$

Bratley's algorithm / Branch and bound algorithm

- based on branch and bound procedure, modified DFS
- build a tree of all permutations
 - in each level l we branch each node into n-l+1 branches, so for the first position we can select from n tasks, for the second from $n-1, \ldots$
 - n! is not a very friendly complexity, so we prune the ones which lead nowhere / are infeasible
- Pruning rules
 - bounding

 if a node exceeds a deadline we can eliminate it along with all its siblings

decomposition

• if we find a delay in the solution, we can consider this node to be so far optimal and don't have to backtrack and consider other paths, as a better solution can provide us only with longer delay and not smaller C_{max}

lower bound

 if we already found a solution check that if we add all the remaining tasks the result is not worse, if it is prune this node

Optimality test

- BRTP (block with release time property)
 - starting from the end, tasks which precede it without breaks form a block
 - the release time for the first one is the earliest among all release times in the block $r_1 \leq r_i \ orall i$
 - it is a sufficient condition, not necessary

• ILP formulation for $1|prec|\sum w_j C_j$

- we are trying to minimize waiting times for all the tasks, and we have precedence, which means that certain tasks have to occur before other tasks
- NP-hard

LP formulation

- variables:
 - ullet x_{ij} : binary matrix, where $x_{ij}=1$ iff T_i precedes T_j or i=j
 - e_{ij} : binary matrix, where $e_{ij}=1$ iff there is directed edge from T_i to T_j in the precedence graph G or i=j, encodes the precedence relations
- constraints:
 - precedence
 - $x_{ij} \geq e_{ij}$
 - either i before j or vice versa
 - $\bullet \ \ x_{ij} + x_{ji} = 1$
 - forbid inactivity and cycles in the task graph

- between three nodes there has to be at least one connection
- ullet $1 \leq x_{ij} + x_{jk} + x_{ki} \leq 2$
- criterion:
 - ullet $C_j = \sum_{i=1}^n p_i \cdot x_{ij}$
 - completion time of task j is the sum of processing times of all tasks that precede it (assuming there are no gaps, which there wont be as we do not have release times for this task)
 - $w_j \cdot C_j = \sum_{i=1}^n p_i \cdot x_{ij} \cdot w_j$
 - add the corresponding weight to the equation
 - $ullet J = \sum_{j=1}^n w_j \cdot C_j = \sum_{j=1}^n \sum_{i=1}^n p_i \cdot x_{ij} \cdot w_j$
 - finally we can sum it across all j
 - $\min_x J(x)$
- ullet Branch and bound with LP bounding for $1|prec|\sum w_jC_j$
 - Tree algorithm similar to Bradley B&B
 - We can bound the branching by using the previously formulated ILP and relaxing it to LP, where we allow $0 \le x \le 1$, which doesn't make sense from the prospective of solving the task, however LP will give us better or equal solution than the ILP will, which we can use in the B&B algorithm
 - EXAMPLE: we already have a solution with obj. value 100 and arrive at a node with obj. value 50, we run the LP and it gives us 101, seeing as even with an optimistic estimate we are over the best solution we found so far we prune this node

Lateness minimalization

- $1||L_{max}$: earliest due date first rule
 - sort the tasks according to their deadlines
 - time $O(n \log n)$
 - Proof of optimality
 - we have an optimal schedule S^{A} from oracle and S^{EDD} schedule
 - if $S^{EDD} \neq S^A$ it implies, that there exist two tasks a and b in the S^A , where $d(a) \leq d(b)$, and b is immediately followed by

- by swapping these we cannot increase the L_{max}
- with finite amount of swaps we obtain $S^{A^\prime} = S^{EDD}$
- $1|r_j|L_{max}$: similar to $1|r_j,\widetilde{d}_j|C_{max}$ and thus NP-hard
- Horn's algorithm

lateness minimalization on one resource, earliest deadline first, recompute when new task is released or a task is finished

- can be used to solve
 - $1|pmtn, r_j|L_{max}$
 - $ullet \ 1|pmtn,r_j,d_j=\widetilde{d}_j|L_{max}$
- set t₁ as the earliest release date
- if all tasks ready at t_1
 - $t_2=\infty$
- else
 - set of ready tasks \mathcal{T}' with release date t_1
 - out of tasks $t
 otin \mathcal{T}'$ select the earliest release date and set it as t_2
 - out of tasks in \mathcal{T}' those choose the one with earliest due date
 - compute delta as the minimum out of the processing time for the chosen tasks and t_2
 - $\bullet \ \ \delta = \min\{p_{t_1}, t_2\}$
 - if the minimum is t_2 we need to do preemption, e.g. shorten the processing time for task t_1 by time until t_2
 - if we finished the task remove it from the pool
 - else do the entire task t and remove it from the ready pool
 - increment the time by δ and continue
- Chetto, Silly, Bouchentouf / $1|pmtn,prec,r_j,d_j=\widetilde{d}_j|L_{max}$
 - we cannot directly use Horn's algorithm because of the precedence relations, so we modify release dates and deadlines in a way that the precedence relations will be satisfied and the use Horns algorithm
 - Algorithm

 draw a tree of precedence and modify the deadlines and release times

Modify release dates

- ullet for tasks without predecessors set the modified release time $r_j'=r_j$
- select a task T_j for which the release time of all of its predecessors were modified
- set

$$r_j' = \max\{r_j, \max_{(r_i', p_i) \in T_p} \{r_i' + p_i\}\}$$

ullet where T_p is the set of all predecessors of j

Modify deadlines

- for tasks without successors set the modified deadline time as $d_j' = d_j$
- select a task T_j , for which the deadline of all of its successors were modified
- set

$$d_j' = \min\{d_j, \min_{(d_i,p_i) \in T_s} \{d_i' - p_i\}\}$$

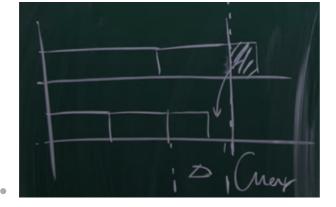
run Horn's algorithm on the updated tasks

"Proof" of correctness

- by using Horn's algorithm on the modified release times and deadlines we satisfy the precedence relations
- release times can be only increased, deadlines can be only shortened, thus this procedure can't introduce anything wrong
- for proof of correctness of Horn's algorithm see Horn's algorithm

Scheduling on parallel identical resources

- McNaughton's Algorithm for $P|pmtn|C_{max}$
 - We have two processors that process our tasks
 - If the task can be performed in parallel we can speed up the tasks by parallelizing the overlap



If not we are limited by the overlap



• compute C_{max}

$$C_{max} = \max\{\max_{i=1:n} p_i, \ rac{1}{R} \sum_i^n p_i\}$$

- where p_i is processing time for the i-th task and R is the number of resources
- draw the diagram with C_{max} and insert the tasks in numerical order, wrap the tasks around the end onto the next available resource (processor)
- List Scheduling (LS), approximation algorithm for $P|prec|C_{max}$
 - for all resources initiate time index t_v
 - initiate start times + allocation variables to 0
 - sort tasks in list L
 - for unsorted list it has approximation factor $r=2-rac{1}{R}$
 - for both $P|prec|C_{max}$ and $P||C_{max}$
 - for all tasks in L
 - choose a resource with lowest t_v (lowest load)
 - subset S with the lowest availability times a_i
 - ullet in case there is no preceding task $a_i=t_v$ for the processor that is processing the task
 - if its predecessors are not finished yet $a_i = \infty$
 - if they are we have to derive it from when its last predecessor ended $a_i = \max_{j \in Pred(T_i)} \{s_j + p_j\}$

remove the task from S which is the first in list L from the list L

Longest processing time first (LPT)

- modification for LS where we sort the list by non-increasing processing times
- reduces the approximation factor to $r=rac{4}{3}-rac{1}{3R}$
- time complexity $O(n \cdot \log(n))$, due to the sorting

• Rothkopf algorithm for $P||C_{max}$

matrix with wave for multiple processors without constraints

- for each task i create binary variable $x_i(t_1,\ldots,t_R)$ where,
 - $i=1,2,\ldots,n$ are task indices
 - $v = 1, 2, \dots, R$ are resource indices
 - $t_v=1,2,\ldots,UB$ are task indices and UB is bound on C_{max} , it can be the sum of processing times
 - $x_i(t_1,\ldots,t_R)=1$ iff tasks T_1,\ldots,T_i can be assigned to resources P_v so they are all occupied in time $<0,t_v>$

Algorithm

- create a search space of dimensionality R imes UB
- for each task $1, \ldots, n$
 - place it into the search space in all places, where it could go
- when we reach n select the place in the tensor with minimal maximum value on each axis and reconstruct the sequence of tasks that lead here
- pseudopolynomial for constant R for changing R it is strongly NP-hard
- time complexity $O(n \cdot UB^R)$
- Muntz&Coffman level algorithm for $P|pmtn, prec|C_{max}$

temporal graph with levels and 3 δ 's

- pick tasks from a list of tasks according to their level
- level of task is the longest path from the task to a task with no successor

- algorithm is exact in case we have 2 resources or in case we have a forest of precedence, for general task we have $r=2-\frac{2}{R}$
- time complexity is $O(n^2)$

Algorithm

- compute the level for all tasks
- while all tasks not finished
 - set Z of tasks with finished predecessors
 - while free resources and |Z| > 0
 - set S of tasks with the highest level
 - for all tasks from S
 - if we have more resources than tasks
 - place tasks on free resources and mark them as full
 - execution speed $\beta = 1$
 - num of free resources h = h |S|
 - if not enough free resources
 - split the tasks evenly among free resources
 - execution speed $\beta = \frac{h}{|S|}$
 - num of free resources h=0
 - and remove the scheduled tasks from Z
 - compute δ , which is
 - time until one task is finished
 - level of a assigned task becomes lower than a level of an unassigned ready task
 - level of task executed at a faster rate β drops below the level of task executed at a slower rate β
 - decrease processing times and levels by $\delta \cdot \beta$
 - jump the time $t=t+\delta$
 - free resources h = R

Project scheduling

- Temporal constraints
 - we have a graph of tasks, where edge values are temporal constraints

- if we have vertices T_i and T_j connected by edge l_{ij} with some value then $s_j \geq s_i + l_{ij}$
- we can also have negative values for the edges, $s_j \geq s_i l_{ij}$ then implies that T_i has to happen at most l_{ij} after task T_j
- we can also use the constraint to specify that two tasks have to start at the same time by setting $l_{ij}=l_{ji}=0 \implies s_i=s_j$
- ILP for $PS1|temp|C_{max}$ one resource
 - Time indexed
 - +
- can be extended for parallel identical processors
- less contraints, more power
- -
- complexity depends on UB
- ullet we consider discretized time with $t\in\mathbb{N}$
- variables
 - $ullet x_{it} \in \{0,1\}$
 - binary matrix tasks × time index
 - UB, upper bound
 - ullet can be the time the latest task ends plus all l_{ij} for all tasks that precede it
- constraints
 - start time can be constructed as

$$s_i = \sum_{t=0}^{UB-1} (t \cdot x_{it})$$

inequalities for temporal constraints

$$\sum_{t=0}^{UB-1}(t\cdot x_{it})+l_{ij}\leq \sum_{t=0}^{UB-1}(t\cdot x_{jt})\quad orall l_{ij}
eq -\infty,\ i
eq j.$$

i.e.

$$s_i + l_{ij} \leq s_j \quad orall l_{ij}
eq -\infty, \ i
eq j$$

• resource constrain, we are on mono-processor so for each task-column in \boldsymbol{x} we have to ensure that tasks will not

overlap (knedlikova podminka)

$$\sum_{i=1}^n \left(\sum_{k=\max(0,t-p_i+1)}^t x_{ik}
ight) \leq 1 \quad orall t \in \{0,\ldots,UB-1\}$$

the task must be scheduled

$$\sum_{t=0}^{UB-1} x_{it} = 1 \quad orall i \in \{1,\ldots,n\}$$

 C max will be equal to the time the completion time of the latest task

$$C_{max} \geq \sum_{t=0}^{UB-1} (t \cdot x_{it}) + p_i$$

i.e.

$$C_{max} \geq s_i + p_i$$

criterion, we are trying to minimalize C max

$$\min C_{max}$$

- Relative order
 - +
- simpler than the first one but requires more constraints
- · complexity doesn't depend on UB
- -
- can not be extended for multiple processors
- variables
 - x_{ij} binary matrix tasks by tasks, where $x_{ij}=1$ if i precedes j
 - start times s_i , which is integer vector
- constraints
 - resource constraint
 - either i follows j or j follows i

$$ullet \ s_i \geq s_j + p_j \ ext{or} \ s_j \geq s_i + p_i$$

• we use big M to formulate this, here M=UB

$$egin{aligned} & x_{ij} \cdot UB + s_i \geq s_j + p_j \ & (1 - x_{ij}) \cdot UB + s_i \geq s_i + p_i \end{aligned}$$

which can be reformulated into one inequality

•
$$p_j \leq s_i - s_j + UB \cdot x_{ij} \leq UB - p_i$$

• $orall i, j \in \{1, \dots, n\}$ and $i < j$

temporal constraint

•
$$s_j \geq s_i + l_{ij} \quad orall l_{ij}
eq -\infty ext{ and } i
eq j$$

- C max
 - $ullet C_{max} \geq s_i + p_i \quad orall i \in \{1,\ldots,n\}$
- criterion
 - $\min C_{max}$
- ILP for $PSm, 1|temp|C_{max}$
 - m represents the number of dedicated resources, e.g. each resource can complete specific tasks
 - Relative order ILP
 - variables
 - x_{ij} binary matrix tasks by tasks
 - constraints
 - resource constraints, again

$$egin{aligned} ullet p_j & \leq s_i - s_j + UB \cdot x_{ij} \leq UB - p_i \ ullet \forall i,j \in \{1,\dots,n\}, i < j \end{aligned}$$

- and $a_i=a_j$ (tasks are on the same resource)
- temporal constraint

•
$$s_j \geq s_i + l_{ij} \quad orall l_{ij}
eq -\infty ext{ and } i
eq j$$

C max

$$ullet \ C_{max} \geq s_i + p_i \quad orall i \in \{1,\ldots,n\}$$

- criterion
 - $\min C_{max}$
- Reduction $PSm, 1|temp|C_{max} o PS1|temp|C_{max}$
 - we have the schedule on several processors and want to execute it on one processor by putting the schedules on each processor behind each other
 - we have upper bound UB on each processor, so we just stack it $m \cdot UB$ onto the one processor

• a_i is the order of the processor sections on the one processor, e.g. the first processor runs in section [0, UB] the second one in $[UB, 2 \cdot UB]$, third in $[2 \cdot UB, 3 \cdot UB]$, etc.

Algorithm

- add dummy vertices (tasks)
 - T_0 which precedes all tasks
 - T_{n+1} which follows after all tasks
 - both with $p_0 = p_{n+1} = 0$
- new start times: $s_i' = s_i + (a_i 1) \cdot UB$
- if we plug the new start times into temporal constraints $s_i + l_{ij} \leq s_j$ we get
- new temporal constraints: $l'_{ij} = l_{ij} + (a_j a_i) \cdot UB$
- Decide whether the new instance is schedulable
- Modeling with temporal constraints
- Cycles and relative time windows
- Otázky z minulých let
 - mono-processor, b&b bratley
 - temporal constraints
 - time indexed ILP
 - Rothkopf (2-processors)
 - Horn algorithm
 - Převod PSm, 1 |temp| Cmax na PS1 |temp| Cmax a určení zda je daná instance rozvrhnutelná
 - Muntz&Coffman level algorithm for P|pmtn,prec|C_max

Constraint programming

- Constraint satisfaction problem
 - Defined by the following sets
 - $X = \{x_1, \dots, x_n\}$ set of variables, e.g. a cell in sudoku
 - $D = \{D_1, \dots, D_n\}$ set of domain variables
 - $D_i = \{v_1, \dots, v_k\}$ is a set of all possible values of x_i
 - e.g. in sudoku the set $\{1,\ldots,9\}$

- $C = \{C_1, \dots, C_n\}$ set of constraints
 - e.g. in sudoku
 - ullet constraint on rows $x_{ij}
 eq x_i orall i$
 - cols $x_{ij}
 eq x_{ij} orall j$
 - cells ...
- CP is a decision problem

Solving CP with search and propagation

- Basic branch and bound with DFS
- One level for each variable
- Number of branches is given by $|D_i|$, e.g. one branch for each value
- Run a DFS but eliminate a branch if it makes some constraints-breaking choice

Arc consistency

- consider binary CSP, where constraint is a binary relation
 - e.g. each constraint considers only 2 domains
- binary CSP can be represented by a digraph G
 - where nodes are variables
 - if there is a constraint between x_i, x_j connect (x_i, x_j) and (x_j, x_i) by oriented arcs
- Arc (x_i,x_j) is Arc Consistent (AC) iff for each value $a\in D_i$ there exists a value $b\in D_j$ such that the assignment $x_i=a,\,x_j=b$ meets all binary constraints for the variables x_i , x_j
- If all arcs are consistent then the entire CSP is consistent

Arc consistency AC-3 Algorithm

- Add all relations to a queue
- Revise each relation
 - Revise procedure for edge (x_i,x_j)
 - ullet erase all values $a\in D_i$ for which there is not a consistent value $b\in D_j$
 - If we repaired inconsistencies we have to add relations we could have broken by this back to the queue again
 - Possible broken relations are relations which lead to this domain and do not belong to the revised domain

Global constraints

- some constraints can be formulated more efficiently if we consider a global structure of the problem
- for example $x_i \neq x_j, \ \forall i \neq j$ can be formulated by an all different constraint which uses a matching algorithm in a bipartite graph

Otázky z minulých let

- Hranová konzistence (AC-3 algoritmus)
- search and propagation algorithm