2 Dynamic Programming

- In what follows, we introduce the technique Dynamic Programming
- It generalizes the basic principles of recursion
- Basic ideas of recursive algorithms
  - The entire problem is separated into smaller predefined subproblems
  - These predetermined smaller subproblems are solved separately
  - Subsequently, the resulting partial solutions are combined to a complete solution to the original problem
  - This generation of a complete solution is determined by a recursive formula
- Typical examples of recursive algorithms
  - Quicksort
  - Recursive version of binary search
  - Recursive version of merge sort

2.1 Basic attributes of Dynamic Programming

- Dynamic Programming generalizes the principles of recursive algorithms
- Here, the particular subproblems that are needed to construct a solution to the larger original problem are unknown
- Therefore, in order to solve the original problem, all smaller subproblems of a specific size have to be solved and all possible combinations (following a recursive formula) have to be assessed while the best one is implemented
- However, if a generated combination leads to an infeasible solution, this alternative is penalized by infinite costs
- Therefore, this alternative is excluded from further considerations

Basic attributes of Dynamic Programming

- Thus, in various applications, the derived subproblems may overlap and, therefore, parts have to be solved multiple times during the optimization process
- This leads to wasted computing time
- A counterstrategy is to explicitly enumerate the distinct subproblems and solve them in the right order
- Moreover, the process can be accelerated by explicitly integrating domination rules and/or bounds fathoming existing partial solutions. These extensions are frequently denoted as bounded Dynamic Programming approaches

2.2 A DP approach for the Knapsack Problem

- In what follows, we introduce a first example of a very simple but somehow interesting Dynamic Programming approach
- It solves the well-known Knapsack Problem (KP) to optimality
- However, since this problem is proven to be (binary) NP-hard, we cannot expect a strongly polynomial solution approach
- The Knapsack Problem has many applications as a problem model or deals as an important subproblem for other applications/optimization problems
- First of all, we briefly introduce the KP itself

2.2.1 The Knapsack Problem

- Given is a set of n items and a fixed capacity limit C
- Each item j has a weight w_j and a price p_j
- Sought: a subset of items which maximizes the total price (objective) and whose sum of weights does not exceed the capacity limit C
- Each item may only be included once
- Consequently, a binary variable x_j indicates whether an item j belongs to the sought subset or not
- The problem is often illustrated by an example where a bag or a knapsack with a fixed maximum weight limit has to be packed with items of different prices and weights
- The 0-1 Knapsack Problem is defined as:

$$\max z = \sum_{j=1}^{n} p_j \cdot x_j \quad \text{s.t.} \quad \sum_{j=1}^{n} w_j \cdot x_j \leq C \quad \wedge \quad x_j \in \{0,1\} \; \forall j = 1,\ldots,n$$

2.2.2 A pseudo-polynomial DP for the KP

- It has been shown that the Knapsack Problem is (binary) NP-hard
- However, its structure allows quite efficient solution approaches
- Among them, the most efficient algorithms for solving the Knapsack Problem are in particular DP approaches
- Applied to the Knapsack Problem, items may be considered iteratively one by one in order to decide about a potential assignment
- Consequently, by defining $f_i(C')$ as the maximal price attainable for assigning the items $1,\ldots,i$, ($i \le n$, $C' \le C$), we calculate

$$f_1(C') = \begin{cases} p_1 & \text{if } 0 \le w_1 \le C' \\ 0 & \text{if } 0 \le C' < w_1 \end{cases}$$

$$f_i(C') = \begin{cases} \max\left\{f_{i-1}(C'), f_{i-1}(C' - w_i) + p_i\right\} & \text{if } C' - w_i \ge 0 \\ f_{i-1}(C') & \text{otherwise} \end{cases}$$

Observations

- Clearly, the algorithm iteratively analyzes whether an assignment of an item is favorable or not
- Consequently, the recursive formula always compares the cases of assigning or not assigning an item
  - Thus, if assigned, the weight of the item accordingly reduces the capacity of the knapsack and provides an additional benefit
  - Otherwise, the element is left outside and the problem is reduced to the optimal assignment of the remaining items 1,…,n-1
- Thus, since we have to calculate $n \cdot C$ function values, the total complexity is restricted by $O(n \cdot C)$

Pseudo-polynomial running time

- A total running time of $O(n \cdot C)$ seems to be moderate and somehow polynomial. Or not?
- What about the factor C?
- It is the capacity of the knapsack and, therefore, not restricted at all
- Unfortunately, it can grow exponentially in the number of elements n
- Note that in the proof of NP-Completeness of the Knapsack Problem, we need an exponentially growing knapsack
- However, if the knapsack is of moderate size (this applies to most applications), the algorithm is quite efficient

A simple example

- Let us consider a small example

$$\text{Maximize } Z = 4 \cdot x_1 + 7 \cdot x_2 + 5 \cdot x_3$$

$$\text{s.t. } 4 \cdot x_1 + 5 \cdot x_2 + 3 \cdot x_3 \leq 10 \wedge x \in \{0,1\}^n$$

- We sort the items according to their efficiency $e_j = \frac{p_j}{w_j}$
- Thus, we obtain the following table

Number of item	Price	Weight	Efficiency
1	4	4	1
2	7	5	7/5
3	5	3	5/3

Observations

- Clearly, the parameter C in the recursive formula gives some potential for optimization
- Since not all partial knapsack capacities are reasonable, i.e., it exists a smaller knapsack obtaining an identical total weight, we may improve the efficiency of the defined calculations by making use of the following scheme
- Specifically, we commence the investigation with an empty knapsack and define each possible assignment by a tuple (p_i, w_i) of total price and total weight
- Thus, we obtain $R^0 = \{(0,0)\}$
- Additionally, we have the following resulting sets

$$t > 0\text{: } R^t = \left\{(p,w) \mid \left((p,w) \in R^{t-1} \vee (p - p_t, w - w_t) \in R^{t-1}\right) \wedge w \leq C\right\}$$

Further observations

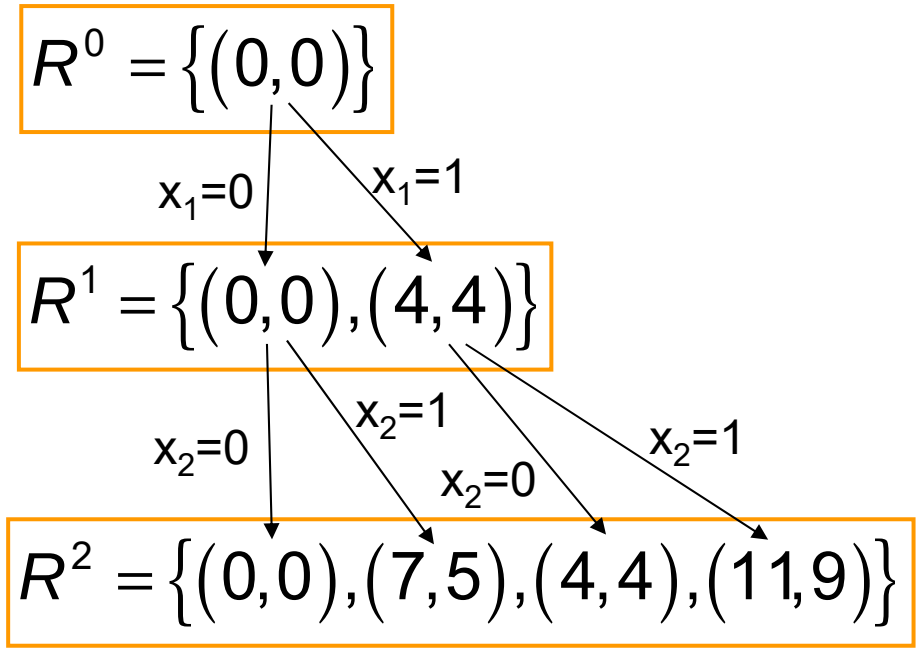
- We may reduce the resulting sets by erasing the dominated tuples
- I.e., a tuple (p,w) is dominated by some tuple (p',w') if it holds that $p \leq p'$ and $w > w'$ or if it holds that $p < p'$ and $w \geq w'$

Applied to our example

- We commence with the initial trivial set

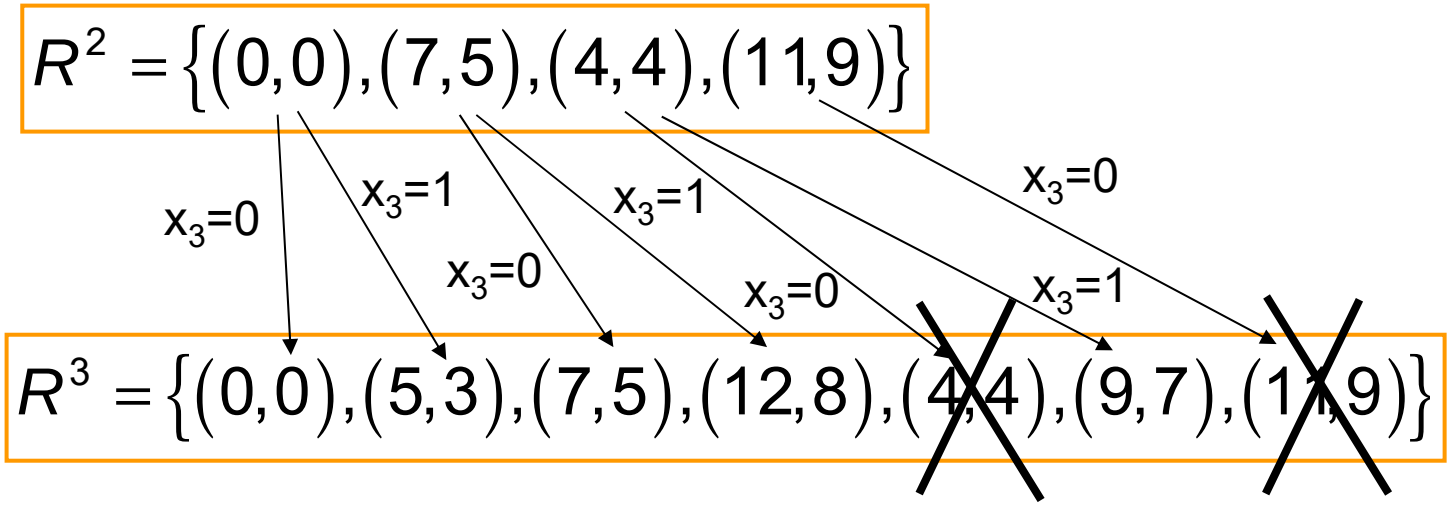
$$R^0 = \{(0,0)\}$$

- Then, we resume with



Applied to our example

- Resume with the set



- Best path

$$(0,0) \rightarrow (0,0) \rightarrow (7,5) \rightarrow (12,8)$$

Possible improvements

- Improvements can be obtained by introducing additional bounds that restrict the number of considered states within the calculation process
- Additionally, dominance criteria may be applied
- Specifically, insights into the problem structure attained by empirical studies reveal that in most cases only a small subset of items around the critical one are necessary to consider
- Simulations with extremely large instances show that…
  - much smaller numbered items are always part of the optimal solution (in these experiments)
  - much larger items are never part of the optimal solution (in these experiments)
- Best bounded DP approaches make widely use of this significant phenomenon

2.3 A DP approach for the aTSP

- In what follows, we introduce a somewhat much more complex DP approach for the asymmetric Traveling Salesman Problem with time windows
- Therefore, we again briefly introduce the problem first

2.3.1 The Traveling Salesman Problem (TSP)

- Most simple model for mapping pure distribution or pick up problems in local traffic
- Problem definition:
  - Given
    - N as the number of customers to be visited
    - Directed weighted graph $G=(V,A,C)$ with
      - $V=\{1,\ldots,N\}$,
      - A set of arcs, and
      - $C=[c_{i,j}]$ the matrix of costs
  - Sought
    - Hamiltonian cycle in G of minimal total costs
    - A Hamiltonian cycle is a cycle passing through each node of G exactly once

The Traveling Salesman Problem

- The TSP has been shown to be an NP-hard problem (Wegener (2005) pp.53-62)
- If costs are symmetric, e.g., the costs for traveling between two locations in the network do not depend on the direction of this travel, a **symmetric TSP** (sTSP) arises; otherwise an **asymmetric TSP** (aTSP) is considered
- If the triangle inequality is always fulfilled or Euclidean distances are used, the problem is termed as the **Euclidean TSP** or the **TSP with the triangle inequality**

Mathematical problem definition

- In the following, we give an exact mathematical definition of the TSP that corresponds directly to a possible solution mapping
- Subsequently, we give the definition for the **asymmetric** case
- Later on in this course, we will also consider the special case of a symmetric TSP. Due to its simplified structure, it allows for the application of specific methods

Asymmetric case: Parameters and variables

Parameters

N : Number of nodes (customers), $V = \{1,...,N\}$

$c_{i,j}\ (1 \le i; j \le N (i \ne j))$: Costs for using the arc (i, j)

Variables

$x_{i,j}\ (1 \le i \le N; 1 \le j \le N (i \ne j))$: Binary variable that is one if and only if the Salesman Path uses the arc (i, j)

Asymmetric case: Restrictions

1. $\forall i \in \{1,...,N\} : \sum_{j \neq i} x_{i,j} = 1 \wedge \sum_{j \neq i} x_{j,i} = 1$ (Each node has two neighbors)

2. For all proper subsets $S \subset \{1,...,N\}$ with $2 \leq |S| \leq \left\lfloor \frac{N}{2} \right\rfloor : \sum_{i \in S} \sum_{j \in V-S} x_{i,j} \geq 1$

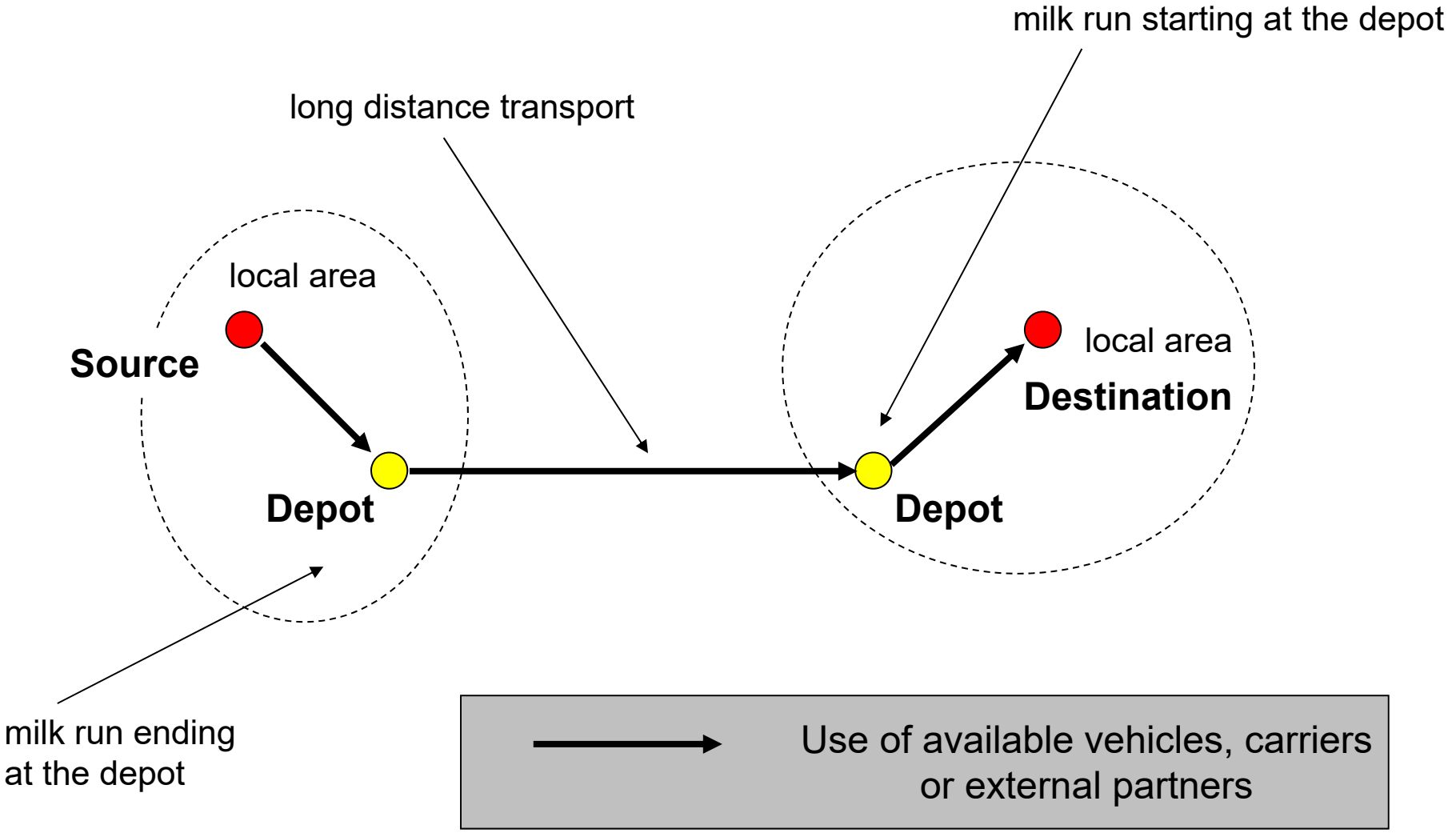
(Subcycle prevention)

3. $\forall i, j \in \{1,...,N\} : 0 \leq x_{i,j} \leq 1$ 4. $\forall i, j \in \{1,...,N\} : x_{i,j}$ is an integer

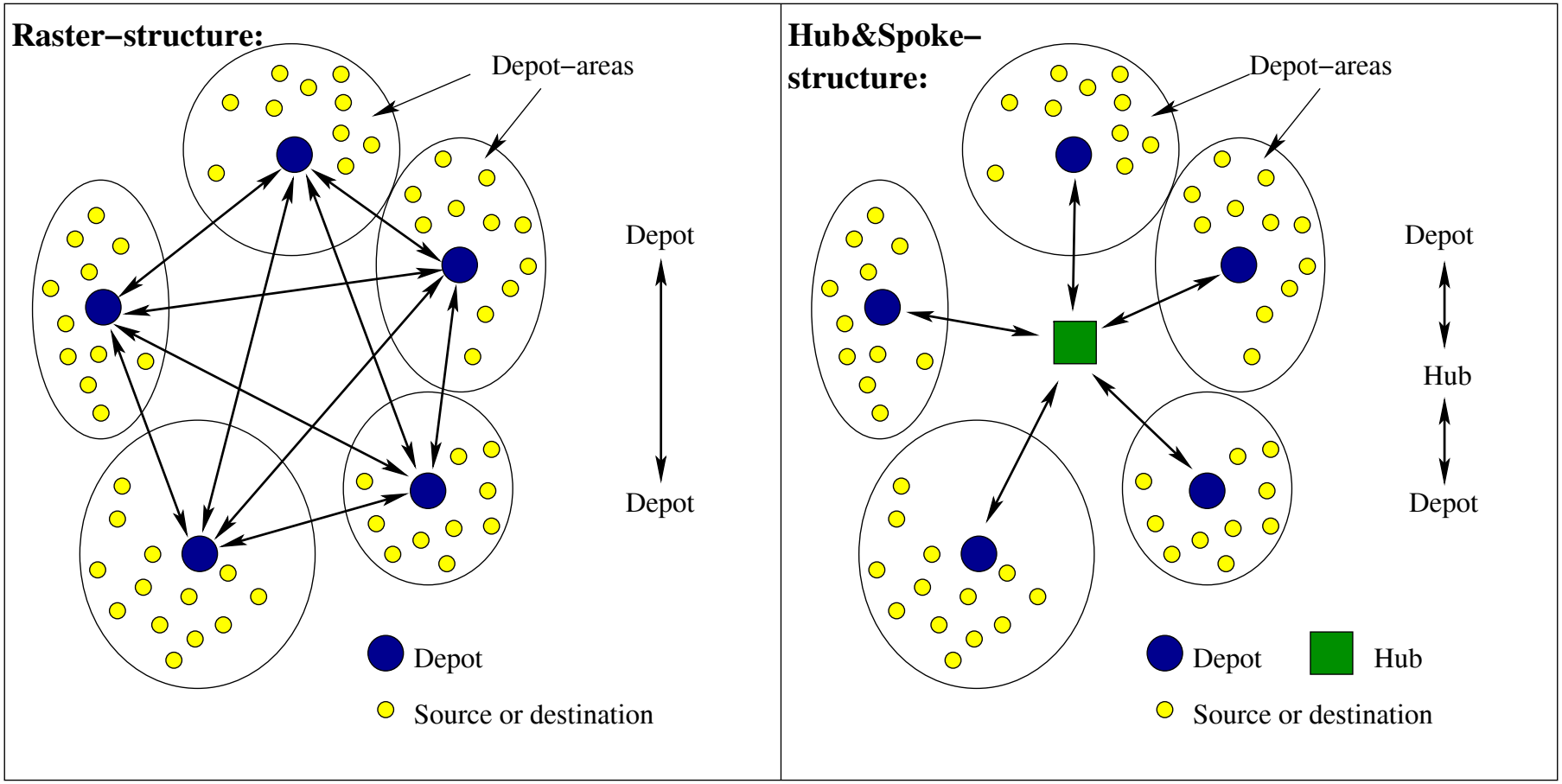
Asymmetric case: Objective function

$$\text{Minimize } F\left(x_{i,j}\Big|_{1\le i,j\le N}\right) = \sum_{i=1}^{N}\sum_{j=1(j\neq i)}^{N} c_{i,j}\cdot x_{i,j}$$

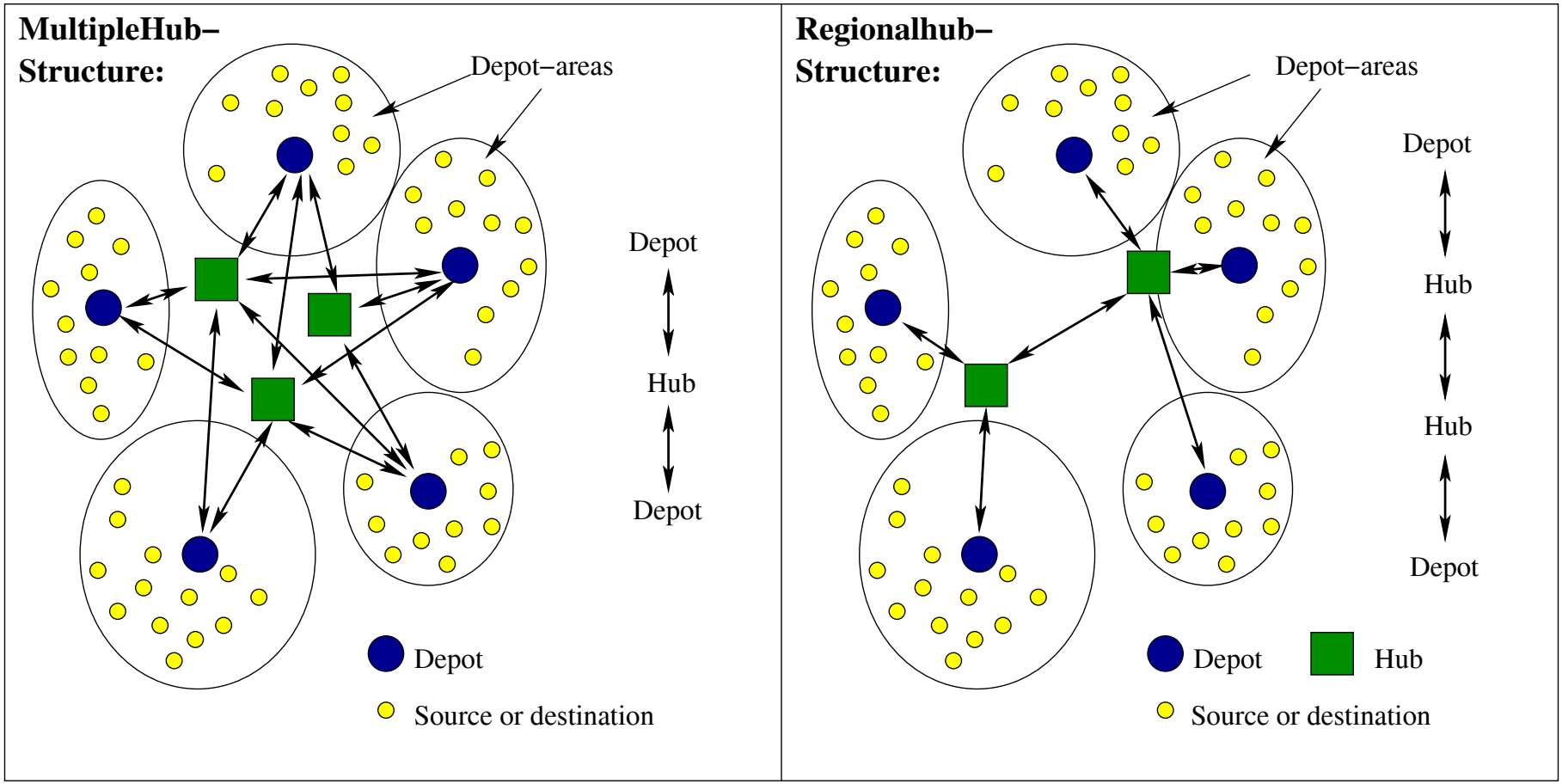
Application of the TSP – planning milk runs



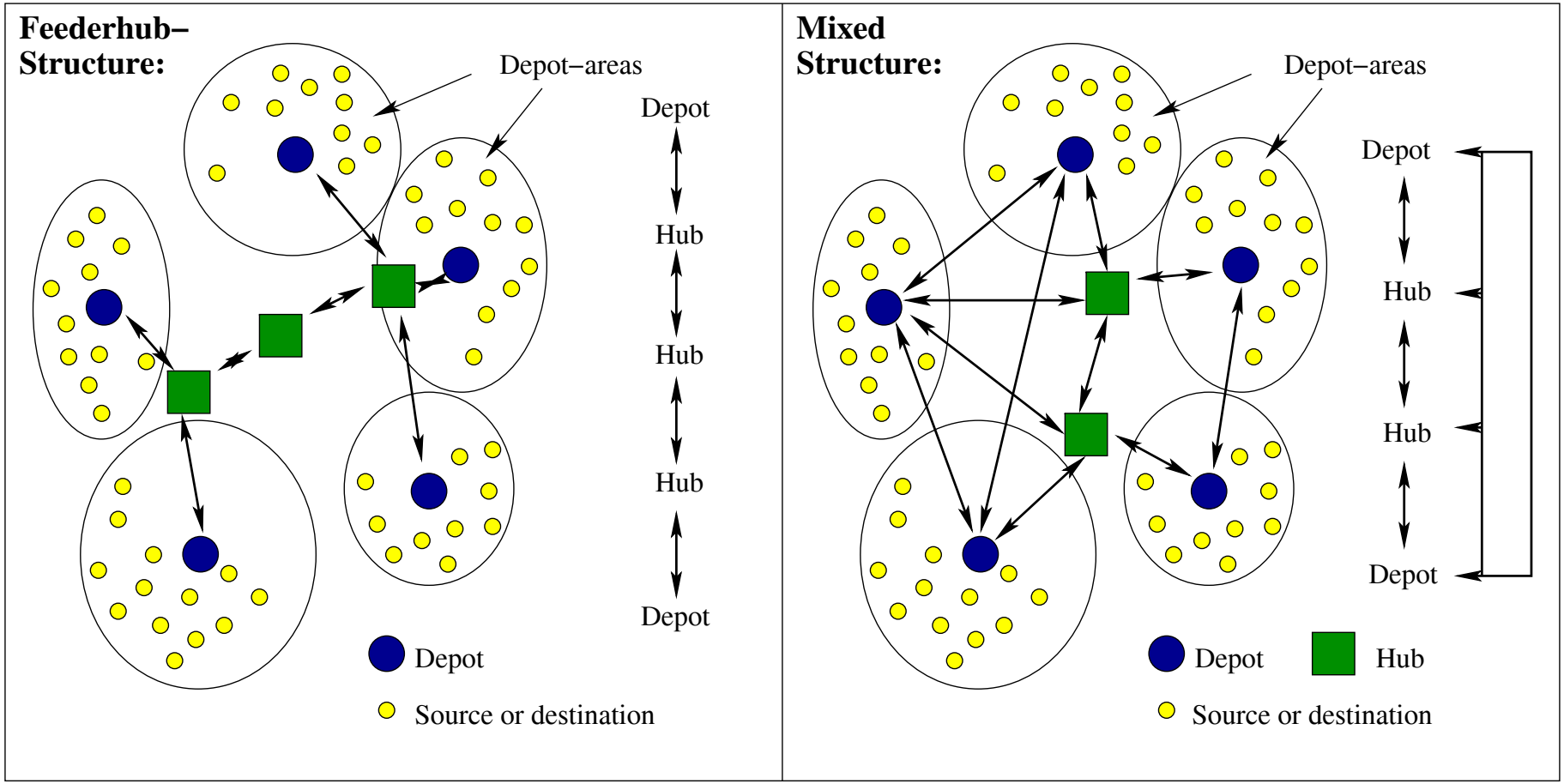
Excursus: Transportation networks I



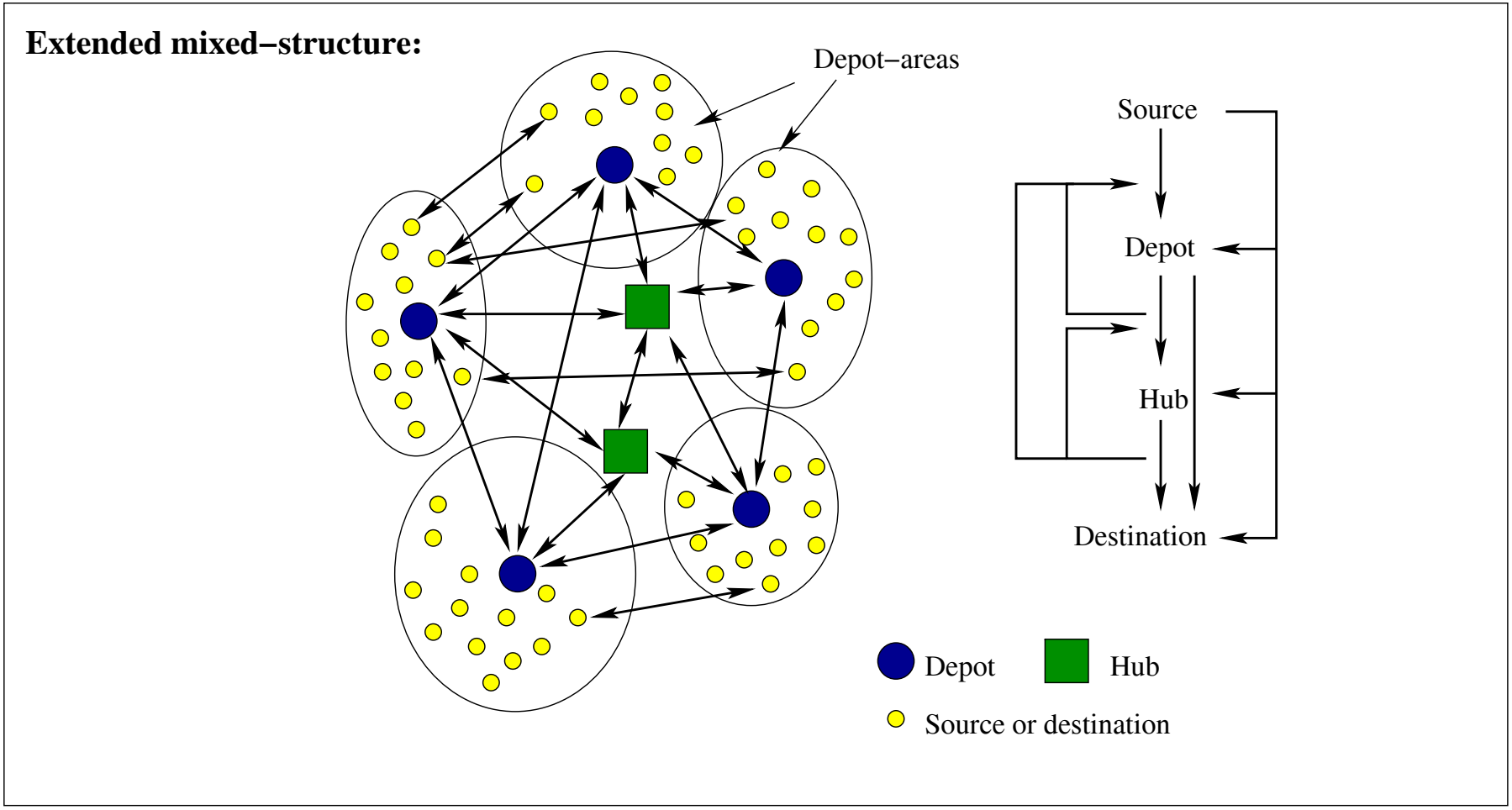
Excursus: Transportation networks II



Excursus: Transportation networks III



Excursus: Transportation network IV



2.3.2 A DP approach for the TSPTW

- In what follows, we introduce a sophisticated exact solution approach to the Traveling Salesman Problem
- This approach was originally proposed by Dumas et al. (1995)
- It considers the TSPTW, i.e., the Traveling Salesman Problem with time windows
  - Here, each customer comes along with a time window that restricts delivery time
  - The time windows $[a_i, b_i]$ are hard, i.e., delivery time at customer i is required to be in the continuous interval $[a_i, b_i]$
  - Additionally, there is a service time s_i at each customer i
- The following solution algorithm bases on a Dynamic Programming concept

2.3.2.1 Notations

- Basic definitions

$G = (N, A)$: Network

$N = \{1, 2, ..., n\}$: Set of customers

$A = \{(i, j) \in (N \times N) \mid a_i + s_i + t_{i,j} \leq b_j\}$

$N' = N - \{n\} = \{1, 2, ..., n-2, n-1\}$

A path in G is defined as a node sequence $i_1, ..., i_k$ such that each arc $(i_j, i_{j+1}) \in A$

Notations

- In what follows, we introduce a somewhat modified notation
- This notation is used in order to introduce the Dynamic Programming approach

Parameters

$t_{i,j}$: Travel time from customer i to j

s_i : Service time at customer i

$[a_i, b_i]$: Time window at customer i

Variables

t_i : Point in time when service starts at customer i

- We define for a path going from customer i to customer j

$$t_j = \begin{cases} \max\{t_i + t_{i,j} + s_i, a_j\} & \text{if } t_i + s_i + t_{i,j} \le b_j \\ \infty & \text{otherwise} \end{cases}$$

Observation

- If a customer is visited too early, waiting time is incurred at this location
- This results from the lower bound a_j in the calculation of t_j
- Note that the considered problem is somewhat modified
  - It seeks the finding of a tour from 1 to n, but not a cyclical tour
  - However, the original problem can be mapped by doubling note 1, i.e., new node n+1 is equal to node 1
  - Then, n is set to n+1

2.3.2.2 Dynamic Programming formulation

- We introduce *F(S,j,t)*
  - Least cost of a path starting at node 1, passing through every node of *S* exactly once, and ending at node *j* at time *t* or earlier
  - We assume that node *j* belongs to set *S*
  - Additionally, *S* is a subset of set *N'*
  - Servicing node *j* is therefore possible at time *t*

Recursive equation

- The function *F(S,j,t)* can be computed by making use of the following equation

$$\forall S \subseteq N' : \forall j \in S : \forall t, \text{ with } a_j \le t \le b_j :$$

$$F(S,j,t) = \min_{(i,j)\in A \wedge i \in S-\{j\}} \left\{ F\left(S-\{j\}, i, t'\right) + c_{i,j} \,\middle|\, t \ge t' + s_i + t_{i,j} \wedge a_i \le t' \le b_i \right\}$$

- We initialize this calculation by

$$\forall j \in \{2,\ldots,n-1\} :$$

$$F(\{1,j\}, j, t) = \begin{cases} c_{1,j} & \text{if } (1,j) \in A \text{ and } \max\{a_1 + s_1 + t_{1,j}, a_j\} \le t \le b_j \\ \infty & \text{otherwise} \end{cases}$$

Determination of optimal solution

- The optimal solution to an instance of the TSPTW can be identified by

$$\min_{(i,n)\in A} \min_{a_i \le t \le b_i} \left\{ F\left(N', i, t\right) + c_{i,n} \,\middle|\, t \le b_n - t_{i,n} - s_i \right\}$$

Observations

- Clearly, the Dynamic Programming approach generates shortest paths in a forward manner
- Specifically, in step $s=1,\ldots,n-1$, a path of length s is constructed
- In each step, a shortest path is sought on a state graph whose nodes are the states $F(S,i,t)$
- Note that the number of states $F(S,i,t)$ $(a_i \leq t \leq b_i)$ is countable if all travel times as well as a_i and b_i are integers
- Among the states, we only conserve the Pareto optimal states
- Note that for two given states $F(S,i,t_1)$ and $F(S,i,t_2)$, the second state can be eliminated if $t_1 \leq t_2$ and $F(S,i,t_1) \leq F(S,i,t_2)$

Cognition

- The function *F(S,i,t)* is stepwise decreasing as a function of *t* over the interval $[a_i, b_i]$
- Clearly, after eliminating the dominated states, the remaining states can be ordered by increasing time and decreasing cost value
- Thus, we obtain the two-dimensional labels *(t,F(S,i,t))* of *(S,i)*
- Therefore, we introduce *FIRST(S,i)* as the first, i.e., lowest, time value of that ordered list of non dominated states
- Thus, *FIRST(S,i)* represents the fastest feasible arrival at node *i* by visiting all other nodes of set S beforehand

2.3.2.3 Post feasibility tests

- The complexity of the described approach is mainly driven by the determination of suitable sets *S*
- Thus, the number of states to be explored grows exponentially
- Consequently, it is very promising to introduce rules and tests in order to eliminate infeasible states, i.e., states that cannot be extended to a complete feasible state
- Hence, these rules are applied in order to eliminate partial paths that do not comply with necessary orderings resulting from existing time window restrictions
- In order to measure earliest arrival times and latest departure times, we introduce the abbreviations EAT and LDT

EAT(i,j), LDT(i,j)

- *EAT(i,j)*
  - Determines the earliest arrival time at node *j* if node *i* is visited before
  - Clearly, this value can be computed by solving a Shortest Path Problem from node *i*
  - Specifically, the path starts at time a_i and satisfies the time window restrictions at all nodes from node *i* to node *j*
- *LDT(i,j)*
  - Determines the latest departure time from node *i* such that t_j is feasible
  - This can be computed by solving a constrained Shortest Path Problem, where the path starts at node *j* at time b_j and uses the reverse arc direction

Additional remarks on EAT(i,j), LDT(i,j)

- Note that the values *EAT(i,j)* and *LDT(i,j)* can be overestimated or underestimated, respectively, by using an unconstrained time path. Since this is much faster to compute in practice, it is frequently applied
- Additionally, if travel times satisfy the triangle inequality, no shortest paths need to be computed. Hence, in this case, *EAT(i,j)* can be replaced by $a_i + s_i + t_{i,j}$

BEFORE(j)

- In what follows, we compute for a node *j* the set of nodes *BEFORE(j)*
- Within this set, there are all nodes that have to be executed before node *j*
- Specifically, it is checked whether the preceding visit of node *j* prohibits a timely service of node *k*
- Consequently, we can compute *BEFORE(j)* as follows

$$BEFORE(j) = \{k \in N \mid EAT(j,k) > b_k\}$$

- Again, if the triangle inequality holds for travel times, $EAT(j,k) > b_k$ can be replaced by $a_j + s_j + t_{j,k} > b_k$

Feasible extensions

- In what follows, we say that a state *(S,i,t)* can be feasibly extended towards *j* if it holds:

$$t + s_i + t_{i,j} \leq b_j$$

- I.e., it holds that:

State $\left(S \cup \{j\}, j, \max\{a_j, t + s_i + t_{i,j}\}\right)$ can be created if it holds:

$t + s_i + t_{i,j} \leq b_j$

Post feasibility test 1

- Given the states *(S,i,t)* for all $a_i \le t \le b_i$
- If the smallest time value to begin service at node *i* is greater than the latest feasible departure time toward *j*, i.e., for all *j* it holds that:

$$FIRST(S,i) + s_i > \min_{j \notin S} LDT(i,j)$$

- Then the states *(S,i,t)* for all $a_i \le t \le b_i$ can be fathomed (deleted) since they do not admit feasible extensions towards any node

Test 1

- This global test eliminates the states *(S,i,t)* for all *t* by only examining the earliest time to begin service at node *i*
- If, for this time, a feasible arrival at any other node is impossible, then all the states *(S,i,t)* are eliminated
- In this case, the states to be treated next are those with a new ending node $i' \in S \backslash \{i\}$ but an identical set *S*
- If, finally, all ending nodes in *S* are explored, a new set *S* is examined

Post feasibility test 2

- Given the states *(S,i,t)* for all $a_i \le t \le b_i$ and given node *j* (*j* does not belong to *S*)
- Additionally, we have *(i,j)* out of *A*
- If

$$BEFORE(j) \not\subset S$$

- then no feasible extension exists towards *j*
- Consequently, node *j* cannot be a successor of *i*
- Thus, different successors have to be tested instead

Post feasibility test 3

Given the state (S,i,t) for a fixed $t, a_i \le t \le b_i$ and given node $j, j \notin S$. Additionally, $(i,j) \in A$.

If (S,i,t) can be extended toward j, i.e., $t + s_i \le LDT(i,j)$, but cannot be extended further to some k, $k \notin S, (i,k) \in A$, i.e., $t + s_i + t_{i,j} + s_j > LDT(j,k)$, then j cannot succeed i for (S,i,t) and the states $(S,i,t'), t' \ge t$ are not extended towards j.

- If the new label $\left(S \cup \{j\}, j, \max\{a_j, t + s_i + t_{i,j}\}\right)$

  is not created, a new node l is considered because no other labels can be feasible for time values greater than t

2.3.2.4 Computational experiments

- The algorithm was coded in C
- Experiments were conducted on a Hewlett-Packard workstation HP9000/730
- By making use of specific preprocessing rules, the arc set A was reduced considerably (Langevin et al. (1990))
- Time windows were also reduced by applying the rules proposed by Desrochers et al. (1992)
- Specific set of problems was tested

  It consists of symmetric Euclidean problems (cf. Langevin et al. (1990))

Set of problems

- Customer coordinates are uniformly distributed between 0 and 50 and travel times equal distances
- Time windows are generated around the times to begin service at each customer of a second nearest neighbor TSP tour
- Each side of a time window is generated as a uniform random variable in the interval [0,w/2], where w=20, 40, 60, 80, and 100
- Clearly, for a given problem size, problem difficulty increases with the time windows' width
  - Consequently, the number of overlapping time windows increases
  - Hence, the number of predetermined node sequences decreases significantly

Complexity of problem instances

Ensuring tight time windows is a hard job. Therefore, tight time windows make the problem hard to solve to optimality

…?!
I believe it is just the opposite.
Tight time windows reduce the feasible solution space

Complexity of problem instances

- Complexity increases significantly with problem size because the geographical area remains constant
- Hence, as *n* increases, the density of the points in this area increases
- In turn, this decreases the ability of the time windows to reduce the number of possible tours
- Whenever time windows start overlapping, exponential complexity prevails and increases the number of tours to be explored significantly
- In what follows, we provide the generated results according to problem sizes (number of customers) and time window widths

Table I
Computational Results for the **TSPTW**

n	w	$\lvert A' \rvert$	Init	Opt	$\lvert S \rvert$	$\lvert (S, i) \rvert$	$\lvert (S, i, t) \rvert$	CPU	Log
20	20	34.2	375.0	361.2	2.8	4.4	4.4	0.02	−1.63
20	40	72.2	349.8	316.0	7.0	14.8	19.6	0.05	−1.33
20	60	115.2	374.4	309.8	14.8	33.8	35.6	0.14	−0.85
20	80	143.8	373.6	311.0	22.4	54.0	64.0	0.23	−0.63
20	100	207.0	373.0	275.2	86.2	244.8	291.4	1.27	0.10
40	20	121.8	526.4	486.6	5.4	9.6	16.0	0.08	−1.12
40	40	230.2	521.0	461.0	14.0	33.0	55.0	0.24	−0.61
40	60	404.2	507.6	416.4	154.6	428.2	554.0	4.37	0.64
40	80	488.8	508.4	399.8	268.2	740.6	890.4	7.52	0.88
40	100	640.8	486.6	377.0	636.6	2644.4	5298.4	31.40	1.50
60	20	226.0	632.8	581.6	7.8	15.2	21.8	0.15	−0.81
60	40	462.8	662.6	590.2	38.6	103.2	145.8	0.89	−0.05
60	60	670.6	684.8	560.0	243.6	626.8	1344.4	6.84	0.83
60	80	987.4	652.6	508.0	1151.8	3526.0	8414.2	46.62	1.67
60	100	1219.8	649.6	514.8	2897.8	8465.8	20846.4	199.84	2.30
80	20	362.6	753.8	676.6	11.0	28.2	49.8	0.35	−0.46
80	40	755.6	730.4	630.0	77.0	198.0	359.0	2.68	0.43
80	60	1149.6	732.6	606.4	843.8	3232.8	7716.8	55.32	1.74
80	80	1490.8	735.8	593.8	3179.4	10449.2	16419.0	220.29	2.34
100	20	510.0	826.4	757.6	14.6	33.4	45.0	0.62	−0.21
100	40	1051.2	830.6	701.8	143.2	442.6	1059.4	7.40	0.87
100	60	1731.4	828.4	696.6	1589.6	4432.2	6804.8	107.95	2.03
150	20	975.8	982.2	868.4	47.4	137.0	326.8	2.44	0.39
150	40	2128.0	996.4	834.8	1484.8	5857.4	20371.0	115.86	2.06
150	60	2953.6	996.6	805.0	4144.6	13158.0	26351.0	462.97	2.67
200	20	1655.6	1143.6	1009.0	60.8	195.8	480.4	6.65	0.82
200	40	3351.2	1163.8	984.2	2380.6	8391.4	14240.0	251.41	2.40

n: the number of nodes;
w: the width of the time windows;
$\lvert A' \rvert$: the arc size of the reduced network;
Init: the average over five problems of the initial solutions obtained by a second nearest neighbor heuristic for the TSP;
Opt: the average over five problems of the optimal solutions for the **TSPTW** (for $n = 100$ and $w = 60$, the average is only over three problems as the memory limitation of 50 Mb was exceeded for the other two problems);
$\lvert S \rvert$, $\lvert (S, i) \rvert$ and $\lvert (S, i, t) \rvert$: the maximum number of labels created;
CPU: the CPU time in seconds on a Hewlett-Packard workstation (HP9000/730, 76 mips, 22 M flops);
log: the logarithm in base 10 of the CPU time.

Direct cognitions

- The presented algorithm was able to solve problems with up to 200 nodes and fairly wide time windows
- As assumed, the CPU time increases with time window width and with problem size
- For narrow window widths, the complexity does not increase exponentially. This allows for solving larger sized problems in reasonable time
- For instance, a 250 node problem with w=20 was solved in less than 10 seconds
- Using the logarithm in base 10 of the CPU time (see Table I), it becomes obvious for a given problem size that the behavior of the algorithm is exponential

Impact of the post feasibility tests

TABLE II

The Impact of Each Test on the Behavior of the Algorithm ($n = 60$)

w =		All Tests	Test 1 Removed	Test 2 Removed	Test 3 Removed
20	$\|(S, i, t)\|$	21.8	21.8	3716.4	37.4
	CPU	0.15	0.15	6.20	0.17
40	$\|(S, i, t)\|$	145.8	145.8	18520.5	229.0
	CPU	0.91	0.91	49.23	0.98
60	$\|(S, i, t)\|$	1344.4	1346.2	*	3864.8
	CPU	6.84	7.42	*	9.54

Conclusions

- As illustrated by table II, the impact of the post feasibility tests was examined additionally
- Clearly, test 2 is the most beneficial one
  - Its absence increases the number of examined states significantly
  - Thus, for problem size n=60, instances with window width w=60 cannot be solved in reasonable time
- Test 1 is the least powerful because much of its work was already performed by reducing the width of the time windows in the preprocessing phase
- For the generation of further problem instances, we fixed w at 60 (fairly wide time windows are generated)
- The number of common arcs between the second nearest neighbor TSP tour and the optimal tour stayed at approximately 20% even for 800 nodes problems

Constant point density problems

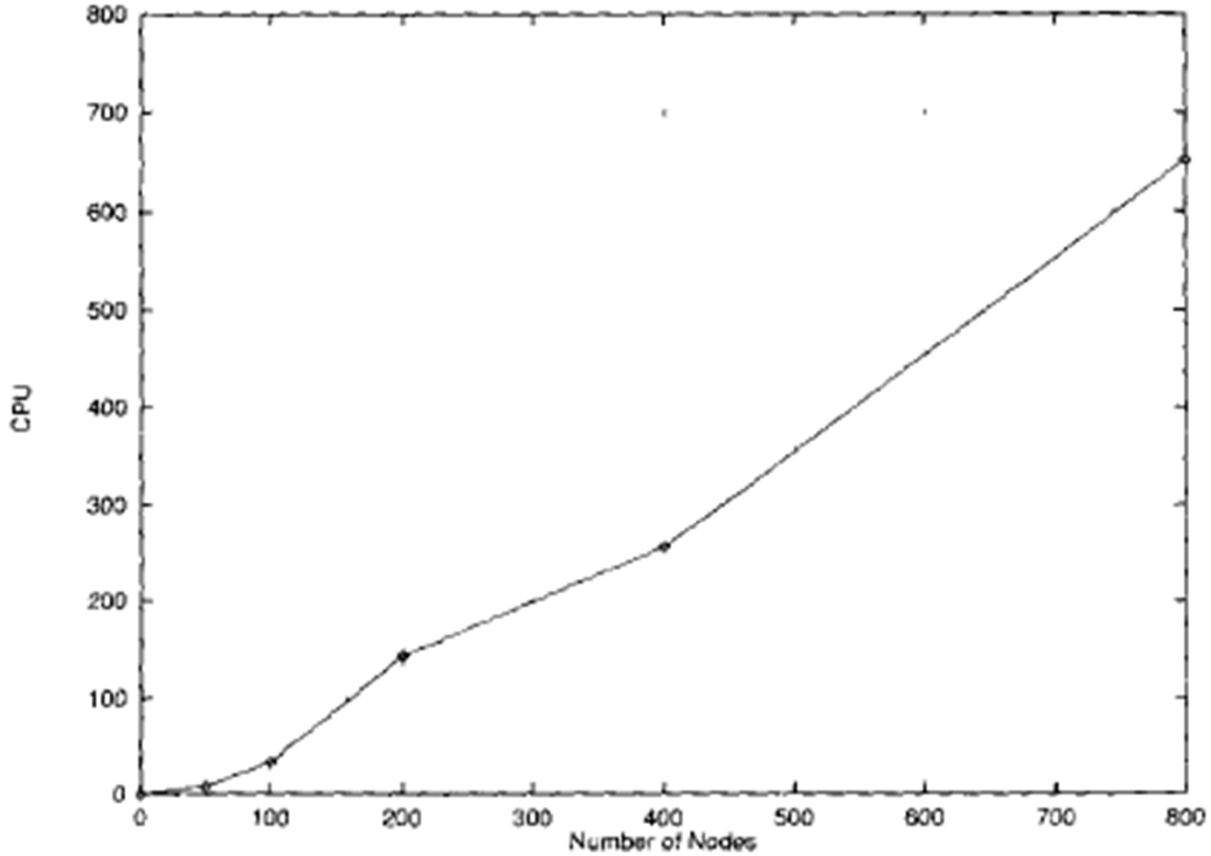


Figure 1. Computational behavior of the algorithm on constant point density problems.

Results

- Owing to the post feasibility rules, problems of size 800 could be solved in approximately 650 seconds
- However, larger size problems face memory limitations

2.3.2.5 A small example

- We consider the following simple example:

$$C = \begin{pmatrix} \infty & 4 & 7 \\ 8 & \infty & 6 \\ 2 & 1 & \infty \end{pmatrix}$$

$$T = \begin{pmatrix} \infty & 1 & 6 \\ 3 & \infty & 8 \\ 6 & 5 & \infty \end{pmatrix}$$

i	s_i	a_i	b_i
1	0	0	100
2	2	3	5
3	4	10	15

Matrix C: $c_{i,j}$:= Cost for traveling from node *i* to node *j*

Matrix T: $t_{i,j}$:= Travel time for traveling from node *i* to node *j*

Nodes start at index 1

Task and start of algorithm

- The TSPTW problem starts at customer/node 1
- Calculate the optimal tour and give the costs as well as the duration of the tour

- Problem starts and ends at node 1
- Customer 1 is doubled to customer 4
  → N = {1,2,3,4}, N‘={1,2,3},

$$\forall i \in \{1,\ldots,3\} : c_{i,4} = c_{i,1}, t_{i,4} = t_{i,1},$$

$$EAT(i,4) = EAT(i,1), LDT(i,4) = LDT(i,1)$$

Calculation of EAT(i,j), LDT(i,j), BEFORE(j)

i \ j	1	2	3
EAT(i,j)			
1		1	6
2	8		13
3	20	19	

LDT(i,j)	1	2	3
1		4	9
2	97		7
3	94	0	

Note: The LDT(i,1)-values are not required and, therefore, not computed.

	1	2	3
BEFORE(j)	{}	{}	{2}

Iteration 1: s=1

- Generate the states *(S,i)* of all sets *S* with $|S| = 2$

i

s	*(S,i)*	2	3
	{1,2}	(3,4)	
	{1,3}		(10,7)

$(t,F(S,i,t)),\ t=3=\max\{a_2, a_1+s_1+t_{1,2}\}=\max\{3,1\}$

- Keep only pareto-optimal states:

(S,i)	2	3
{1,2}	(3,4)	
{1,3}		(10,7)

Post feasibility tests

- Test 1: Partial solution ({1,3}, 3, 10) is not extendable towards j=2 → remove this state from the available states
- Test 2 and Test 3 do not eliminate any further states
- Resulting available partial solution:

(S,i)	2	3
{1,2}	(3,4)	

Iteration 2: s=2

- Extend every available partial solution to states with $|S|=3$
- Available set of states only consists of the state ({1,2}, (3,4)). Extend this state by adding node 3:

(S,i)	3
{1,2,3}	(13,10), (14,10), (15,10)

(t,F(S,i,t)), with $t=13=\max\{a_3, 3+s_2+t_{2,3}\}=\max\{10,3+2+8\}$

- Keep only pareto-optimal states:

(S,i)	3
{1,2,3}	(13,10)

Example – Calculation of optimal solution

- All nodes of *N'* have been added to the available states
- Calculating the optimal solution by the formula

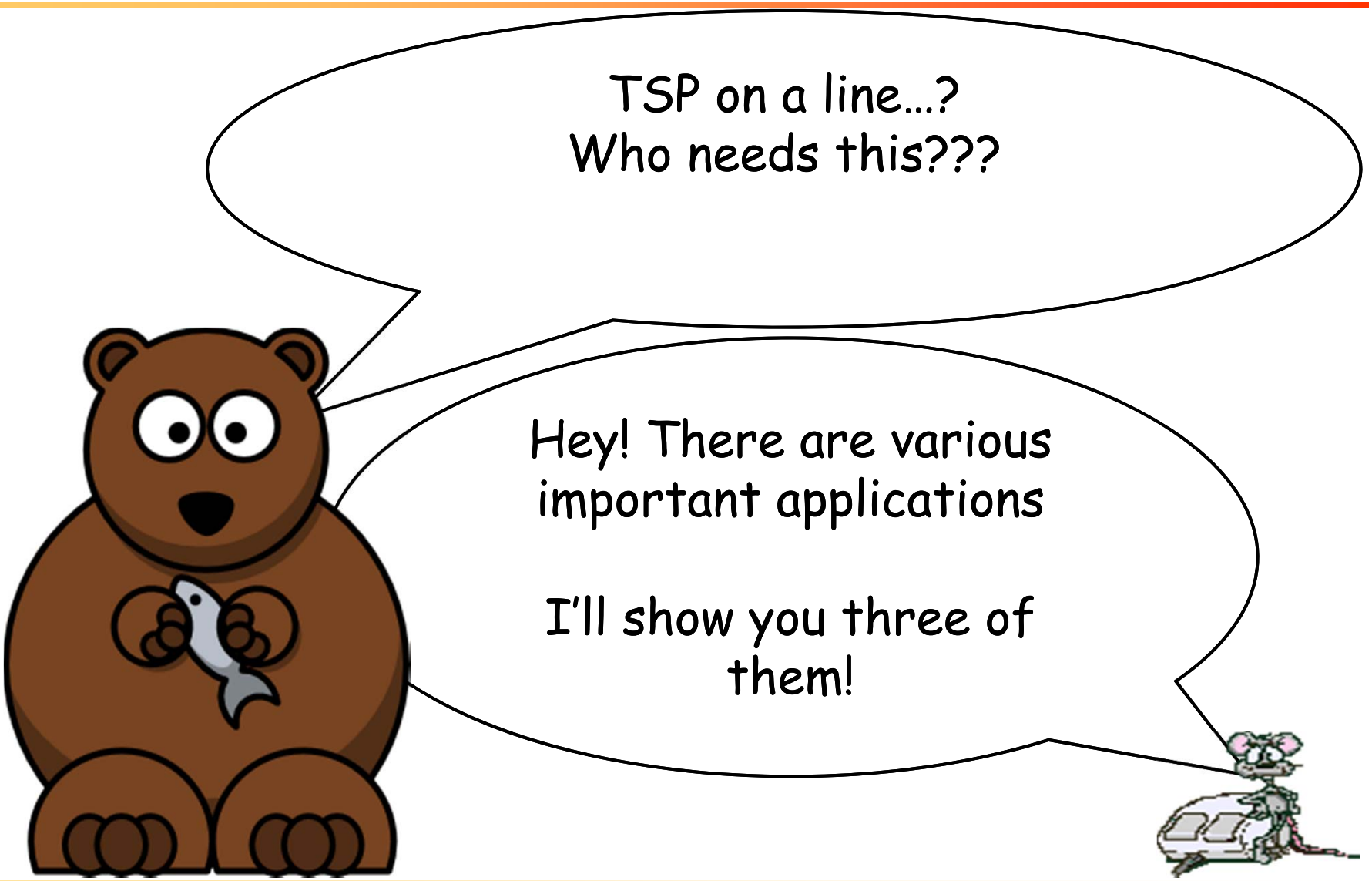
$$\min_{(i,n)\in A} \min_{a_i \le t \le b_i} \left\{ F(N', i, t) + c_{i,n} \,\middle|\, t \le b_n - t_{i,n} - s_i \right\}$$

- Only one state available → Calculate costs of optimal solution by $F(\{1,2,3\},3,13) + c_{3,4}$ $= 10 + 2 = 12$
- Duration: $13 + s_3 + t_{3,4} =$ $13 + s_3 + t_{3,1} = 13 + 4 + 6 = 23$
- Optimal solution: 1-2-3-1, Costs: 12, Duration: 23

2.4 Exactly solving the Line TSP by DP

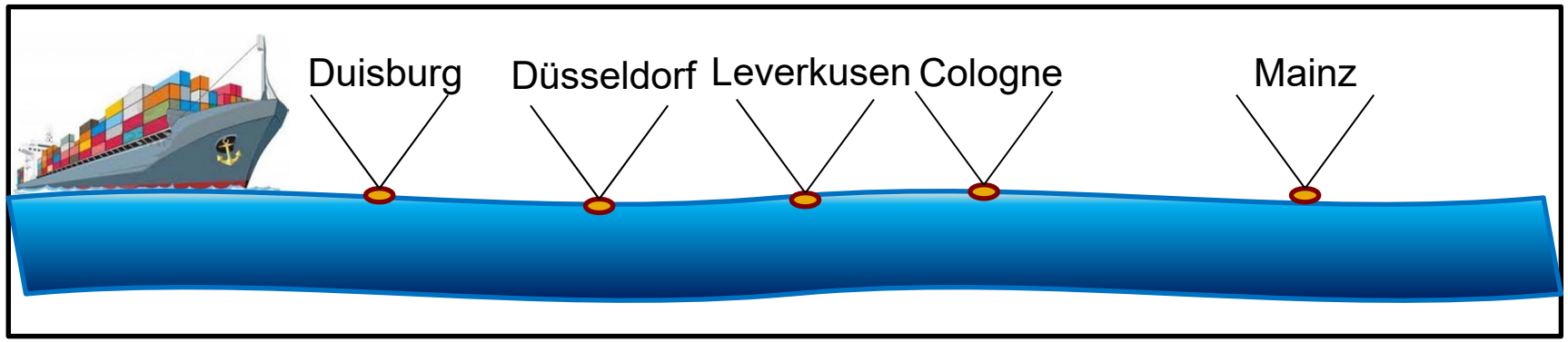
- A specific case of the Traveling Salesman Problem occurs whenever all customers are located along a straight line and the starting position x* of the salesman is also on this line
- E.g., there is a single road that connects all these customers and along which supply is executed
- Therefore, each customer i is located at position x_i and it holds: $c_{i,j}=|x_j-x_i|$
- We additionally introduce time windows $[r_i,d_i]$ into the problem definition. Specifically, r_i determines the release time and d_i a due date at the location of customer i
- Moreover, at each location i, a handling or service time h_i is given
- Objective function is defined as the minimization of the maximum completion time (i.e., the makespan)

Line TSP


TSP on a line…?
Who needs this???
Hey! There are various important applications
I'll show you three of them!

Applications: Inland ship along the Rhine

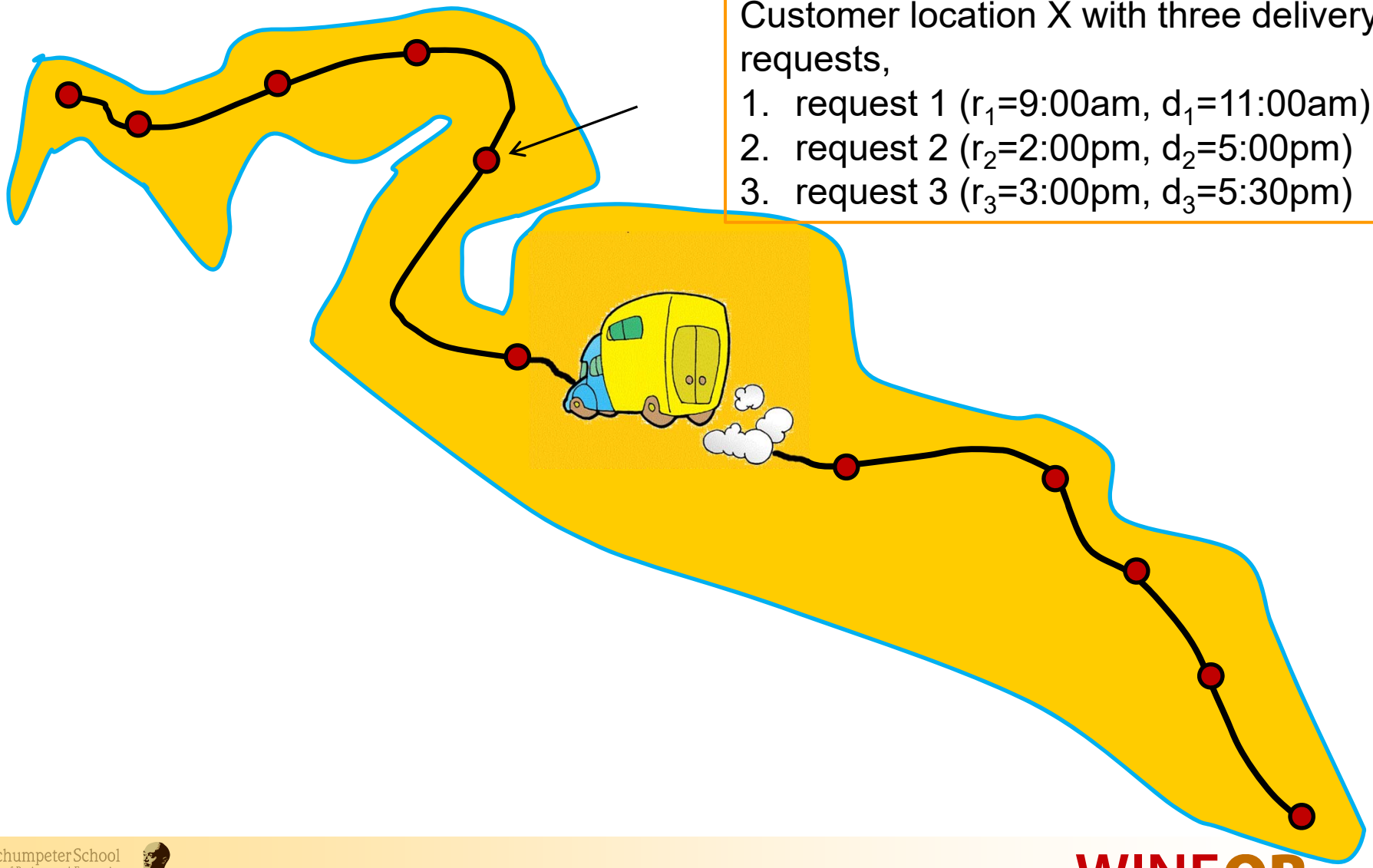
- Cargo ships have to supply inland ports along the Rhine
- Individual time restrictions have to be obeyed at each port
  - There may be ports that permit a delivery before the release date
  - Latest date of supply (hard time window)
- Different direction-dependent travel speeds (in flow/against flow direction)
- Minimization of the tour length fulfilling all due dates



In flow direction: faster transportation possible

Against flow direction: slower transportation

Applications: Delivery along a coast line

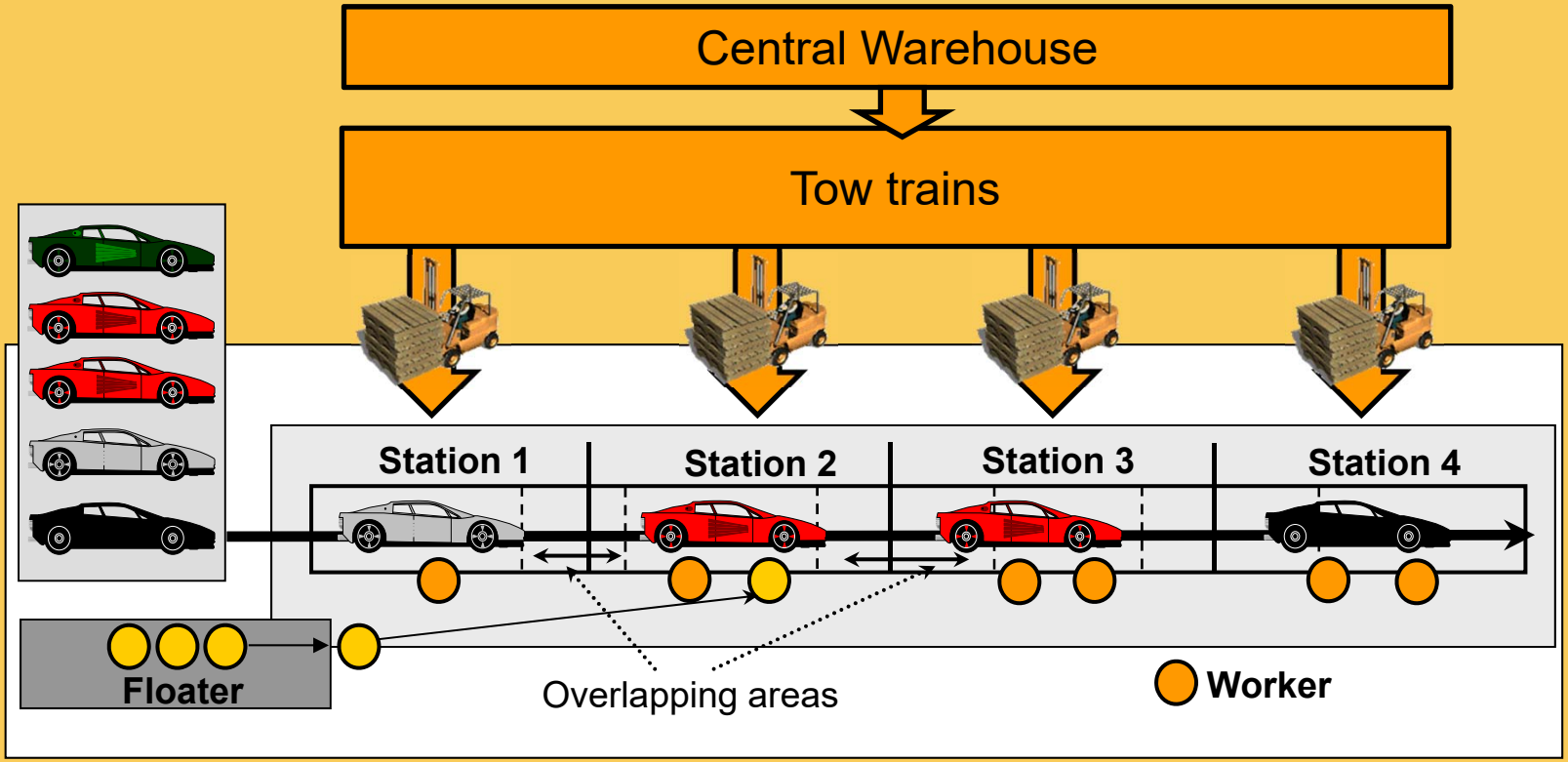


Customer location X with three delivery requests,

1. request 1 (r_1=9:00am, d_1=11:00am)
2. request 2 (r_2=2:00pm, d_2=5:00pm)
3. request 3 (r_3=3:00pm, d_3=5:30pm)

JiT supply of mixed-model assembly lines

- Stations are supplied by specific tow trains
- Tight time window constraints to be obeyed at the stations arise from the launched variant sequence



Mixed-model assembly line

A polynomial special case

- In what follows, we consider the special case $r_i=h_i=0$ (i.e., zero release dates and no service or handling time at the delivery locations)
- In that special constellation, we can provide a very efficient solution procedure working in quadratic time complexity, i.e., this problem is well-solvable

Solving the special case of the Line TSP

2.4.1 Theorem

The special case of Line TSPTW with n customers in which it holds $r_i=h_i=0$ for all i can be optimally solved in time $O(n^2)$

Proof of Theorem 2.4.1

- Clearly, since processing times and release dates are zero, it costs nothing to serve a customer when traveling by
- Therefore, it is assumed that each customer is immediately served the first time its location is visited by the vehicle
- Specifically, if the vehicle has visited locations a and b with $a \leq b$, then all jobs whose locations belong to the interval [a,b] have been serviced
- In what follows, we assume that job numbers are sorted according to their locations, i.e., it holds $x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_n$
- Moreover, we assume that the starting position $x^* = x_{i^*}$ coincides with the location of some customer
- Note that such an artificial customer can be generated without causing additional costs

Proof of Theorem 2.4.1

- Moreover, in order to ensure feasibility, we assume that $|x_{i*}-x_i| \leq d_i$ for all i; otherwise the problem is not solvable at all
- We fix some pair of jobs (i and j) with $1 \leq i \leq i^* \leq j \leq n$
- We consider all schedules where job i is timely visited for the first time (i.e., not after the due date) and, before that, all jobs that are located in the interval $[x_i, x_j]$ were visited timely
- Based on these schedules, we introduce two additional abbreviations
  - $V^-(i,j)$

    Earliest point in time where such a constellation is possible
  - $V^+(i,j)$

    Earliest point in time in a schedule where job j is timely visited for the first time and, before that, all jobs that are located in the interval $[x_i, x_j]$ have been visited timely as well
  - Note that if those time points do not exist, these abbreviations are set to infinity

Proof of Theorem 2.4.1

- Observations
  - Clearly, we know $\forall i^* < j : V^+(i^*, j) = x_j - x^*\ (1)$
  - Moreover, it holds: $\forall i < i^* : V^+(i, i^*) = \infty\ (2)$
- (1) results from the fact that we can directly use the direct path to x_j
- (2) directly follows since we start at position i* and cannot serve customer i before
- Furthermore, we conclude $V^+(i^*,i^*)=0$

Proof of Theorem 2.4.1

- Analogously, we can conclude that
  - It holds: $\forall i < i^* : V^-(i, i^*) = x^* - x_i \quad (3)$
  - Moreover, it holds: $\forall j > i^* : V^-(i^*, j) = \infty \quad (4)$
- (3) again results from the fact that we can directly use the direct path to x_i
- (4) directly follows since we start at position i* and cannot serve customer j before
- Furthermore, we obtain $V^-(i^*,i^*)=0$

Additional abbreviations

- We introduce $U^+(i,j)$ as the minimum point in time where we can reach customer j after serving all customers i, i+1, …, j-1
- Thus, we apply the following recursive formula

$$\forall i < i^* < j : U^+(i,j) =$$
$$\min\left\{V^+(i,j-1) + x_j - x_{j-1}, V^-(i,j-1) + x_j - x_i\right\} \quad (5)$$

- Clearly, (5) results from the following cognitions
  - If we want to visit customer j for the first time, we either come from j-1 or
  - we come from customer i
  - Otherwise, we would not have visited all customers i, i+1, …, j-1 before

Additional abbreviations

- Analogously, we introduce $U^-(i,j)$ as the minimum point in time when customer i is reached after we have served all customers i+1, i+2, …, j
- Thus, we apply the following recursive formula

$$\forall i < i^* < j : U^-(i,j) =$$
$$\min\left\{V^-(i+1,j) + x_{i+1} - x_i, V^+(i+1,j) + x_j - x_i\right\} \quad (6)$$

- Clearly, (6) again results from the following cognitions
  - If we want to visit customer i for the first time, we either come from customer j or
  - we come from customer i+1
  - Otherwise, we would not have visited all customers i+1, i+2, …, j-1, j before

Consequence

- Based on these abbreviations, we can introduce the following useful recursions
- Specifically, it holds:

$$\forall i < i^* < j : V^+(i,j) = \begin{cases} U^+(i,j) & \text{if } U^+(i,j) \le d_j \\ \infty & \text{otherwise} \end{cases} \quad (7)$$

- and additionally:

$$\forall i < i^* < j : V^-(i,j) = \begin{cases} U^-(i,j) & \text{if } U^-(i,j) \le d_i \\ \infty & \text{otherwise} \end{cases} \quad (8)$$

Proof of Theorem 2.4.1

- Consequently, by applying these formulas, we can generate $V^+(1,n)$ and $V^-(1,n)$
- Clearly, the minimum of both is the completion time of the optimal tour table
- I.e., $\min\{V^+(1,n), V^-(1,n)\}$ provides the minimal completion time of the vehicle tour
- This provides us with a Dynamic Programming procedure
- Initially, we calculate the values
  - $V^+(i^*, j)$ for $j>i^*$, and
  - $V^-(i, i^*)$ for $i^*>i$
- Subsequently, we can apply the formulas (7) and (8) in order to get the following values
- Finally, $V^+(1,n)$ and $V^-(1,n)$ are computed

Computational effort

- We have to calculate two arrays (V+ and V-) of customer pairs i, j, with i<j
- Thus, we have to generate iteratively $O(n^2)$ values
- The generation of each value requires constant time
- Specifically, we just have to evaluate two different values whose generation again takes constant time
- All in all, we obtain a total running time of $O(n^2)$
- The optimal tour can be obtained by storing a flag for each pair (i,j) that indicates whether the plus or the minus case was favorable

2.4.2 Further results

	Zero processing or handling (service) times	**General processing or handling (service) times**
No release times or deadlines	Trivial case	Trivial case
Release times only	$O(n^2)$	NP-complete Pseudo-polynomial unknown
Deadlines only	$O(n^2)$	NP-complete Pseudo-polynomial unknown
General time windows	Strongly NP-complete	Strongly NP-complete

2.5 Dynamic Programming in Scheduling

- In what follows, we consider a very sophisticated example of a Dynamic Programming approach that once solved an open research question
- *Is it possible to provide a pseudo-polynomial solution approach for finding an optimal schedule for one stage scenarios while minimizing the total sum of tardiness?*
- Fortunately, the answer was yes, but required the understanding of a specifically designed solution approach
- It is based on Dynamic Programming

2.5.1 Job Shop Scheduling – Basics

- In what follows, we consider scheduling problems
- I.e., we state the following problem

  Given

  - M machines or resources, N jobs to be produced
  - Each job comprises a predetermined set of operations to be executed on the resources

  Sought

  - A production sequence of all N jobs for each machine
  - Determination of the timetables
- Consequently, we have to decide about
  - The sequence of the competing jobs on the machines
  - and the resulting timetable

2.5.1.1 Assumptions

- Production program is given
- Lot sizes are given
- Process sequence of each job is given
- Operating times are given
- No operation of the jobs can be processed simultaneously on more than one machine
- Every machine can process at most one job at each point of time
- All N jobs and its data are available (static problem) at the beginning of the planning horizon
- There are never bottlenecks concerning transports and storage
- No maintenance and reparatory activities
- On each machine, setup times are independent of the realized operation sequence

Given and sought

- Given
  - MS: Machine sequence matrix
  - PT: Matrix of the processing times

- Sought
  - JS: Job sequence matrix
  - TT: Timetable planning matrix with

$t^{a}_{m,n}\left(1 \le m \le M; 1 \le n \le N\right)$ Point of time in which the processing of job n starts at machine $m\left[\text{TU}\right]$

2.5.1.2 Mathematical model

Variables

$t^{a}_{m,n}\left(1 \le n \le N; 1 \le m \le M\right)$: see above

$y_{m,n,k}\left(1 \le m \le M; 1 \le n \le N; 1 \le k \le N\right)$: Binary variable defining the sequence of jobs, i.e.,

$$y_{m,n,k} = \begin{cases} 1 & \text{if job } n \text{ is processed on machine } m \text{ before job } k \\ 0 & \text{otherwise} \end{cases}$$

Mathematical model – Restrictions

Machine sequence restrictions

(derived from the matrix MS):

$$\forall m \in \{1,\ldots,M-1\} : \forall n \in \{1,\ldots,N\} : t^{a}_{[m]_n,n} + p_{[m]_n,n} \leq t^{a}_{[m+1]_n,n}$$

$[m]_n$ defines here the index of the machine

that executes the *m*th operation of job *n*

Mathematical model – Restrictions

- In case of the job sequence restrictions, the formulation depends on the structure of the found solution
- But we have to ensure that two jobs are never processed simultaneously on one machine and, therefore, an arbitrary sequence of those jobs has to be realized

Therefore, there are the following two possible cases:

First case $(n \text{ before } k)$:

$$(1) \quad t^a_{m,n} + p_{m,n} \leq t^a_{m,k}$$

Second case $(k \text{ before } n)$:

$$(2) \quad t^a_{m,k} + p_{m,k} \leq t^a_{m,n}$$

$\Rightarrow$ Both possibilities have to be considered in the model!

Mathematical model – Restrictions

Job sequence restrictions (depends on the chosen solution):

$$\forall m \in \{1,\ldots,M-1\} : \forall n,k \in \{1,\ldots,N\} : t^a_{m,n} + p_{m,n} \leq t^a_{m,k} + \left(1 - y_{m,n,k}\right) \cdot C$$

$$\forall m \in \{1,\ldots,M-1\} : \forall n,k \in \{1,\ldots,N\} : t^a_{m,k} + p_{m,k} \leq t^a_{m,n} + y_{m,n,k} \cdot C$$

C defines a number that is larger than each definition of the timetable variables $t^a_{m,n}$, e.g., $C = \sum_{m=1}^{M} \sum_{n=1}^{N} p_{m,n}$

Mathematical model – Domains

$$\forall m \in \{1,\ldots,M\} : \forall n,k \in \{1,\ldots,N\}(n \neq k) : y_{m,n,k} \in \{0,1\}$$

$$\forall m \in \{1,\ldots,M\} : \forall n \in \{1,\ldots,N\} : t^{a}_{m,n} \geq 0$$

2.5.1.3 Objective functions

- The model defined above can be seen as a general starting point for so-called Job Shop Scheduling problems
- It abstains from the definition of a particular objective function but can be extended by a specific application-dependent definition
- In literature, a huge set of different objective functions is proposed. These functions mainly influence the efficiency of applied solution procedures
- In what follows, we will give some examples of well-known objectives

Minimization of total makespan

- This objective function minimizes the duration for producing the total production quantities, i.e., it pursues the minimization of the maximum completion time over all processed jobs

$$\text{Minimize } Z_1 = t_{\max} = \max\left\{ t_{[M]_n,n} \mid n \in \{1,\ldots,N\} \right\}$$

with:

$\forall n \in \{1,\ldots,N\}$: $t_{[M]_n,n}$: Point of time in which the last processing of job n is completed

Minimization of machine waiting times

- Sum of all machine waiting times for all used resources

$$\text{Minimize } Z_2 = \sum_{m=1}^{M} \underbrace{\left(t_{\max} - \sum_{n=1}^{N} p_{m,n} \right)}_{\text{Unused capacity of machine } m}$$

$$= M \cdot t_{\max} - \left(\sum_{m=1}^{M} \sum_{n=1}^{N} p_{m,n} \right)$$

Note:

Since $\left(\sum_{m=1}^{M} \sum_{n=1}^{N} p_{m,n} \right)$ and M are constants, Z_1 and Z_2 are equivalent

Minimization of total completion time

- This objective tries to minimize the total sum of all individual completion times
- Therefore, we compute the sum of completion times over all processed jobs

$$\text{Minimize } Z_3 = \sum_{n=1}^{N} t_{[M]_n,n}$$

This objective is equivalent to the minimization of the sum of waiting times of all jobs

$$\text{Minimize } Z_4 = \sum_{n=1}^{N} \sum_{m=1}^{M} w_{m,n}$$

Minimization of total lateness

- Here, we want to minimize the total lateness over all N jobs to be produced in the considered production system
- Consequently, a compensation between early and late deliveries is no longer possible

$$\text{Minimize } Z_6 = \sum_{n=1}^{N} \max\left\{t_{[M]_n,n} - d_n, 0\right\}$$

with:

d_n : Due date of job n

Minimization of maximum lateness

- By using this objective, we somehow try to balance the lateness in the found solution among the different jobs
- Thus, we try to minimize the maximum lateness of a job in the found solution

$$\text{Minimize } Z_7 = \max\left\{\max\left\{t_{[M]_n,n} - d_n, 0\right\} \mid n \in \{1,\ldots,N\}\right\}$$

with:

d_n : Due date of job n

Min. of sum of weighted completion times

- Here, each job obtains an individual weight rating its completion time in the production system
- Thus, we receive a combined weighted sum of job completion times

$$\text{Minimize } Z_8 = \sum_{n=1}^{N} w_n \cdot t_{[M]_n,n}$$

with: w_n : Weight of product n

2.5.1.4 Schedule classes

- In the following, we introduce some basic terms according to specific types of schedules
- In the scheduling theory, a distinction is frequently made between
  - Sequence,
  - Schedule, and finally
  - Scheduling policy

Basic terms

- Sequence

  Corresponds to a specific permutation of jobs to be processed on a given machine

- Schedule

  Usually corresponds to an allocation of jobs within a more complicated setting of machines that could allow for preemption of jobs by other jobs that are released at later points in time. Comprises timetables

- Scheduling policy

  Often used in stochastic settings; a policy prescribes an appropriate action for any of the states the system may be in. In deterministic cases, usually only sequences or schedules are of importance but can be extended by rule definitions

Non-delay schedules

2.5.1.4.1 Definition:

A feasible schedule is called non-delay if no machine is kept idle when there is an operation available for processing

Non-delay schedules

- These schedules are not allowed to comprise *unforced idleness of machines* in the production process
- Here, in most cases, we consider non-delay schedules since otherwise an improvement possibility seems to be straightforward
- HOWEVER: There may be some special constellations for non-preemptive models where it pays to have periods of unforced idleness. This results from some specific effects of non-delay schedules (c.f. Pinedo, M. (2012) pp.22)

Active schedules

2.5.1.4.2 Definition:

A feasible schedule is called active if no operation can be completed earlier by starting earlier or by changing the process sequence on machines without delaying any other operation

Attributes of active schedules

2.5.1.4.3 Lemma:

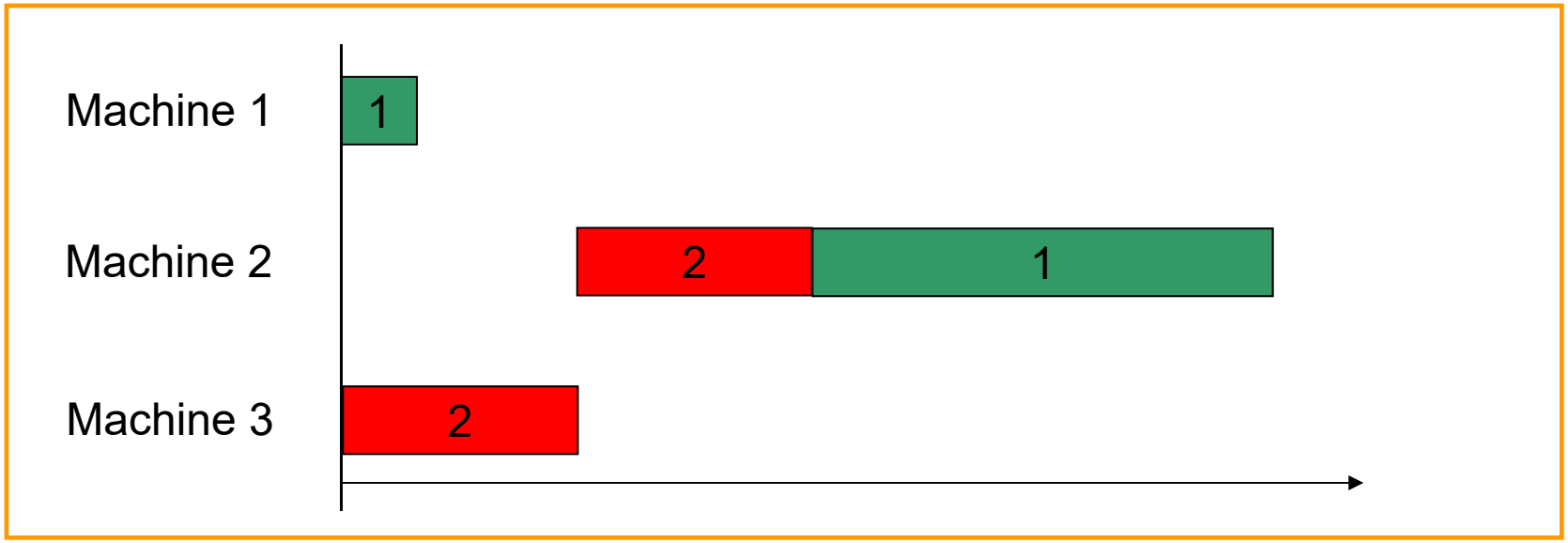
A non-delay schedule is always active

Proof of the Lemma

- Let us assume there is a non-delay schedule that is not active
- Then, we know there is a machine m where shifting an operation of job i to an earlier position at point of time t results in an earlier completion without delaying the other operations
- But, if this is true, we know that during the processing of the schedule on machine m, there is a constellation at point of time t where the considered machine m is idle but can process job i instead
- This contradicts the assumption that the schedule is non-delay

Attributes of active schedules

- Note that the reverse is not necessarily true
- I.e., there are some active schedules that are not non-delay
- Example: Schedule is active but not non-delay



Semi-active schedules

2.5.1.4.4 Definition:

A feasible schedule is called semi-active if no operation can be completed earlier without altering the processing sequence on any of the machines

Consequences

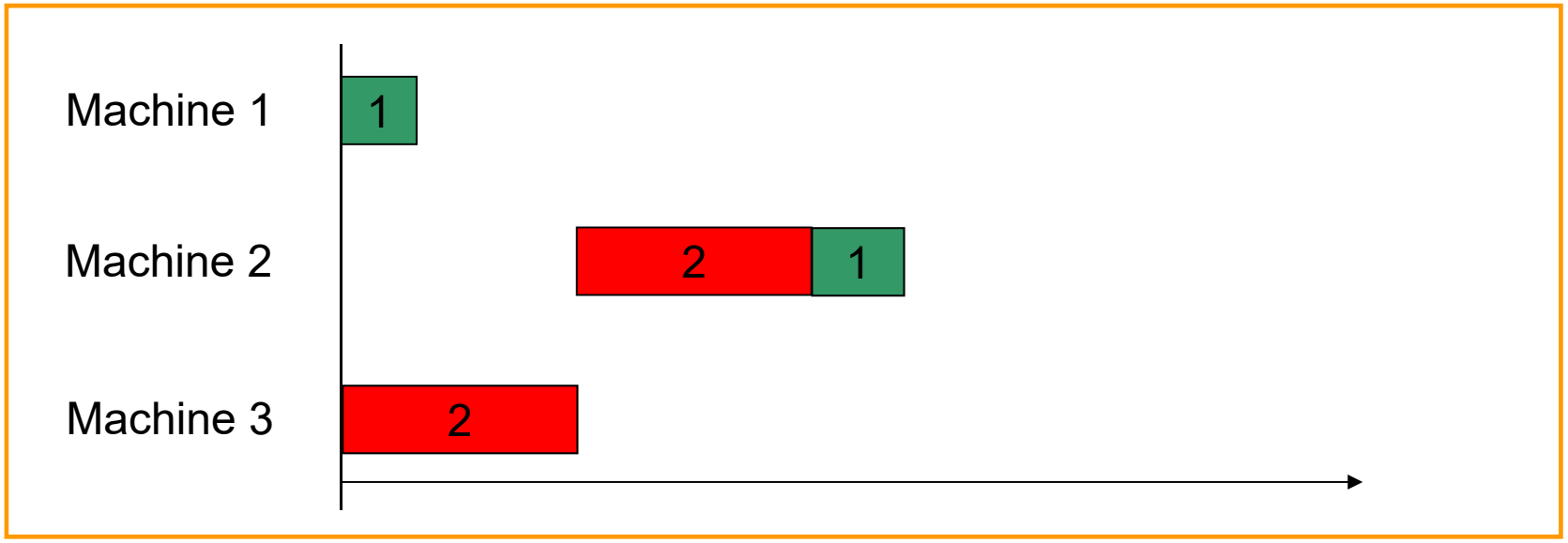
2.5.1.4.5 Lemma:

An active schedule is always semi-active

The proof is trivial and follows immediately from the definition

Attributes of semi-active schedules

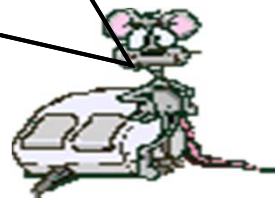
- Note that the reverse is not necessarily true
- I.e., there are some semi-active schedules that are not active
- Example: Schedule is semi-active but not active



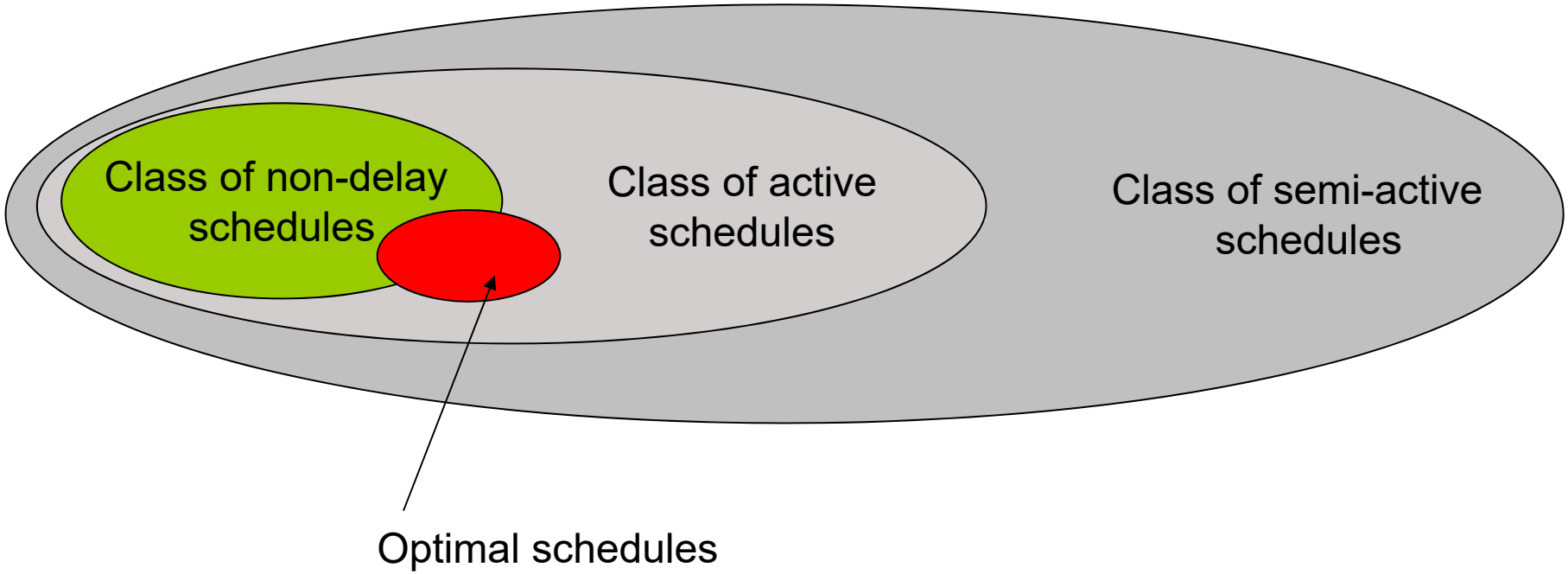
What is the best schedule class?

I hate waiting. Therefore, I always would go for non-delay!

Be careful! Even for monotonous objective functions, waiting can make sense….if it is – for instance - a job with high weight that would be delayed otherwise



Schedule class hierarchy



2.5.2 The considered single-stage problem

- In what follows, we consider a problem that was very challenging and received an enormous amount of attention until its status has been completely clarified in 1990
- It is the single-stage scheduling problem pursuing the minimization of total tardiness
  - N jobs have to be scheduled at a single stage
  - Each job possesses an individual soft due date
  - Objective is the minimization of total tardiness (without weights)
- It has been proven in 1990 by Du and Leung that this problem is already binary NP-hard
- Hence, due to the theory of NP-Completeness, it is conjectured that this rules out the existence of strongly polynomial algorithms
- However, there is a very efficient DP algorithm that works in pseudo-polynomial time (introduced by Lawler (1977))

2.5.3 The DP for the considered problem

- Clearly, by analyzing the problem in detail, it becomes obvious that there is the following general simple dominance criterion applicable (Emmons (1969))

2.5.3.1 Lemma:

If $p_i \leq p_j$ and $d_i \leq d_j$, then there exists an optimal schedule that sequences job *i* before job *j*

Proof:

- The proof is trivial since it follows directly from the cognition that preferring *i* against *j* does not have negative side effects

Proof of dominance criterion

- Specifically, if we have a schedule with the two jobs in opposite sequence, we exchange them
- Consequently, the intermediate completion times of jobs scheduled between the two jobs are not increased
- The completion times of jobs scheduled before behind both jobs are not affected at all
- Due to the fact that job *i* is more urgent than job *j*, we conclude that the exchange cannot increase the total objective value
- This completes the proof

Due date sensitivity of the optimal schedule

- We consider optimal schedules and ask how far we can postpone the due date of a job without affecting the optimality of a solution to the original problem
- For this purpose, we define
  - First instance P' with *n* jobs
    - and processing times $p_1,\ldots,p_n$ as well as due dates $d_1,\ldots,d_n$,
    - Optimal solution S' with maximum completion time $C'(k)$ of job k
  - Second instance P'' with *n* jobs
    - and processing times $p_1,\ldots,p_n$ as well as due dates $d_1,\ldots,max\{d_k,C'(k)\},\ldots,d_n$,
    - Optimal solution S'' with completion time $C''(k)$ of job k

Due date sensitivity of the optimal schedule

2.5.3.2 Lemma:

Every sequence that is optimal for the second instance is also optimal for the first one

Proof:

- We define $V'(S)$ and $V''(S)$ as the total tardiness of a schedule S for the due date structure of the instances P' and P''.
- Let $V'(S')=V''(S')+A_k$ and $V'(S'')=V''(S'')+B_k$ be the total tardiness under the respective instances.
- Clearly, if it holds that $C'(k)\leq d_k$, both sets of due dates are the same and we have identical optimal solutions. Therefore, the proposition follows.

Proof of Lemma 2.5.3.2

- Thus, we assume that it holds $C'(k)>d_k$
- $V'(S')=V''(S')+A_k$ and $V'(S'')=V''(S'')+B_k$
- In this case, we have: $A_k=C'(k)-d_k$
- Moreover, it holds that $B_k=max\{0, min\{C''(k),C'(k)\}-d_k\}$
- Clearly, we conclude the following:
  - $min\{C''(k),C'(k)\}\leq C'(k) \rightarrow B_k\leq A_k$
  - Since S'' is optimal for P'', we obtain $V''(S'')\leq V''(S')$
  - Consequently, we obtain $V'(S'')\leq V'(S')$
- Therefore, since S' is optimal for P', we obtain $V''(S'')=V''(S')$ and S'' is optimal for P'
- This completes the proof

Assumptions

- In what follows, we assume (without loss of generality)
  - All job processing times are different from each other
  - Jobs are renumbered in sequence of non-decreasing due dates, i.e., it holds that $d_1 \leq d_2 \leq \ldots \leq d_n$
  - Moreover, we define $p_k = max\{p_1, p_2, \ldots, p_n\}$
  - Consequently, the kth urgent job has the largest processing time
- Clearly, due to Lemma 2.5.3.1, we know that there is an optimal schedule where all jobs of set $\{1,\ldots,k-1\}$ will be scheduled before job k
- However, what about the remaining jobs?

Consequence

2.5.3.3 Lemma:

There exists an integer δ with $0 \leq \delta \leq n-k$ such that there exists an optimal schedule S in which job k is preceded by all jobs j with $j \leq k+\delta$ and followed by all jobs j with $j > k+\delta$

Proof:

- Let *C'(k)* denote the latest completion time of job *k* in an optimal schedule of the original problem (due dates $d_1,\ldots,d_n$)
- S'' is optimal for the modified problem (with due dates $d_1,\ldots,max\{d_k,C'(k)\},\ldots,d_n$) and satisfies the restriction of Lemma 2.5.3.1

Proof of Lemma 2.5.3.3

- Let C''(k) be the completion time of job k under schedule S''
- Clearly, by applying Lemma 2.5.3.2, schedule S'' is also optimal for the original problem instance P'
- We therefore conclude that $C''(k) \leq max\{C'(k), d_k\}$
- This is true since C'(k) is assumed to be the latest completion time of job k in all optimal schedules for P'
- Moreover, we know that all jobs with a due date later than $max\{C'(k), d_k\}$ are processed after job k (if this would not be the case, we can reassign this job after job k without deteriorating the objective function value)

Proof of Lemma 2.5.3.3

- Clearly, additionally, job k is preceded by all jobs with a due date earlier than $max\{C'(k),d_k\}$ (this results from a direct application of Lemma 2.5.3.1)
- Consequently, we set δ to the maximum integer such that $d_{k+\delta} \leq max\{C'(k),d_k\}$
- This completes the proof

Consequences of findings

And what is won?
No one knows the optimal solutions beforehand. Therefore, the δ value is unknown

This is just the reason why we apply Dynamic Programming. We enumerate all possible values

Consequences of findings

- The Dynamic Programming approach utilizes a procedure that generates an optimal schedule for the set of jobs *1,…,l* with a job k that has the largest processing time
- Due to the results derived above, we know that we have to enumerate possible positions of job k while this job, fortunately, *splits the other jobs into two independent sets*, namely the following job sets with their positions in the corresponding schedule
  - jobs $1,\ldots,k-1,k+1,\ldots,k+\delta$ are processed (in some order) firstly
  - job *k* processed after them and, finally, the
  - jobs $k+\delta+1,\ldots,l$ that are processed in some order lastly
- This leads to the following recursive computation

The recursive formula of the DP

$J(j,I,k) = \{i \mid j \le i \le I \wedge p_i < p_k\}$

$V(J(j,I,k),t)$: Minimal total tardiness for processing the jobs of set $J(j,I,k)$ in an optimal sequence starting at time t

It holds that:

$V(\varnothing,t) = 0$ and $V(\{j\},t) = \max\{0, t + p_j - d_j\}$

$$V(J(j,I,k),t) = \min_{\delta} \left(\begin{array}{l} V(J(j,k'+\delta,k'),t) + \max\left(0, \underbrace{t + p_{k'} + \sum_{j \in J(j,k'+\delta,k')} p_j}_{\text{Completion time of job } k'} - d_{k'}\right) \\ +V\left(J(k'+\delta+1,I,k'), \underbrace{t + p_{k'} + \sum_{j \in J(j,k'+\delta,k')} p_j}_{\text{Completion time of job } k'}\right) \end{array} \right),$$

with $k' \in J(j,I,k)$ such that $p_{k'} = \max\{p_i \mid i \in J(j,I,k)\}$

Optimal solution is obtained by solving: $V(\{1,\ldots,n\},0)$

2.5.4 A simple example

Jobs	1	2	3	4	5
p_j	121	79	147	83	130
d_j	260	266	266	336	337

- We consider an example with 5 jobs that are already sorted according to urgency
- Clearly, we have to solve V({1,2,3,4,5},0)
- For this purpose, we ask for the job with largest processing time
- It is job 3. Hence, jobs 1 and 2 are scheduled before job 3
- We test the positions 3,4,5 for job 3 (i.e., we set δ to 0,1, and 2)
- This leads to the following result

Solving the example

$$V(\{1,2,3,4,5\},0)$$

$$= \min \begin{cases} V(J(1,3,3),0) + \max\{121+79+147-266,0\} + V(J(4,5,3),347) \\ V(J(1,4,3),0) + \max\{121+79+83+147-266,0\} + V(J(5,5,3),430) \\ V(J(1,5,3),0) + \max\{121+79+83+130+147-266,0\} + V(\varnothing,560) \end{cases}$$

$$= \min \begin{cases} V(\{1,2\},0) + 81 + V(\{4,5\},347) \\ V(\{1,2,4\},0) + 164 + V(\{5\},430) \\ V(\{1,2,4,5\},0) + 294 + V(\varnothing,560) \end{cases}$$

- The remaining smaller problems can be solved directly
- Clearly, V({1,2},0)=0 with both sequences (1,2) and (2,1)
- Moreover, V({1,2,4},0)=0 with (1,2,4) or with (2,1,4)
- V({4,5},347)=347+83-336+560-337=317 with the optimal sequence (4,5)

Solving the example

- Moreover, we obtain V({5},430)=430+130-337=223
- Finally, we have V({1,2,4,5},0)=121+79+83+130-337=413-337=76 with the optimal schedules (1,2,4,5) or (2,1,4,5)
- By inserting these values, we obtain

$$V(\{1,2,3,4,5\},0)$$
$$= \min\begin{cases} V(\{1,2\},0)+81+V(\{4,5\},347) \\ V(\{1,2,4\},0)+164+V(\{5\},430) \\ V(\{1,2,4,5\},0)+294+V(\varnothing,560) \end{cases} = \min\begin{cases} 0+81+317 \\ 0+164+223 \\ 76+294+0 \end{cases} = \min\begin{cases} 398 \\ 387 \\ 370 \end{cases}$$

- Therefore, we obtain the optimal schedules (1,2,4,5,3) and (2,1,4,5,3) with total tardiness 370

Complexity of the DP

- The worst case complexity of the approach can be derived directly by estimating the number of occurring problems to be solved
- Each problem is solved in time O(n) (Testing all possible δ-values)
- Moreover, we have $O(n^3)$ sets V(j,l,k) and Σp_j time assignments in the problems to be considered
- Consequently, the number of problems is upper bounded by $O(n^3 \cdot \Sigma p_j)$
- The overall running time is therefore bounded by $O(n^4 \cdot \Sigma p_j)$
- Hence, we have an algorithm with a pseudo-polynomial running time

References of Section 2

- Bock, S.; Klamroth, K.: Minimizing sequence-dependent setup costs in feeding batch processes under due date restrictions. Journal of Scheduling 16: 479-494, 2013.
- Bock, S.: Solving the Traveling Repairman Problem on a line with general processing times and deadlines. European Journal of Operational Research, Vol. 244(3), S.690-703, 2015.
- Desrochers, M.; Desrosiers, J.; Solomon, M.: A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows. Operations Research 40: 342-354, 1992.
- Du, J.; Leung, J.: Minimizing Total Tardiness on One Machine is NP-Hard. Mathematics of Operations Research 15(3): 483-495, 1990.
- Dumas, Y.; Desrosiers, J.; Gelinas, E.; Solomon, M.M.: An optimal Algorithm for the Traveling Salesman Problem with Time Windows. Operations Research 43(2): 367-371, 1995.
- Carey, M.R.; Johnson, D.S.: Two-processor scheduling with start times and deadlines. SIAM Journal on Computing 6: 416-426, 1977.
- Emmons, H. (1969) “One-Machine Sequencing to Minimize Certain Functions of Job Tardiness”. Operations Research, Vol. 17, pp. 701–715.

References of Section 2

- Langevin, A.; Desrochers, M.; Desrosiers, J.; Soumis, F.: A Two-Commodity Flow Formulation for the Traveling Salesman and the Makespan Problems with Time Windows. Working Paper CRT-732, Centre de Recherche sur les Transports, Montreal, Canada, 1990.
- Lawler, E.L. (1977) “A ’Pseudopolynomial’ Time Algorithm for Sequencing Jobs to Minimize Total Tardiness”. Annals of Discrete Mathematics, Vol. 1, pp. 331-342.
- Potts, C.N.; van Wassenhove, L.N. (1982) “A Decomposition Algorithm for the Single Machine Total Tardiness Problem”, Operations Research Letters, Vol. 1, pp. 177–181.
- Potts, C.N.; van Wassenhove, L.N. (1983) “An Algorithm for Single Machine Sequencing with Deadlines to Minimize Total Weighted Completion Time”. European Journal of Operational Research, Vol. 12, pp. 379–387.

References of Section 2

- Potts, C.N.; van Wassenhove, L.N. (1987) “Dynamic Programming and Decomposition Approaches for the Single Machine Total Tardiness Problem”, European Journal of Operational Research, Vol. 32, pp. 405–414.
- Pinedo, M.L.: Scheduling: Theory, Algorithms and Systems. 4th edition, Prentice Hall, New Jersey, 2012. (ISBN-10: 1-4614-1986-0)
- Psaraftis, H.N.; Solomon, M.M.; Magnanti, T.L.; Kim, T.-U.: Routing and scheduling on a shoreline with release times. Management Science 36(2): 212-223, 1990.
- Tsitsiklis, J.N.: Special Cases of Traveling Salesman and Repairman Problems with Time Windows. Networks 22: 263-282, 1992.
- Vahrenkamp, R.; Mattfeld, D.C.: Logistiknetzwerke - Modelle für Standortwahl und Tourenplanung. Gabler, Wiesbaden, 2007. (ISBN 978-3-8349-0541-3)
- Wegener, I.: Theoretische Informatik. Eine algorithmenorientierte Einführung. In German. 3rd edition, Teubner, Wiesbaden, 2005. (ISBN 3.8351-0033-5)