Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Recoverable Robust Optimization for (some) combinatorial problems

Marjan van den Akker

joint work with

Han Hoogeveen, Paul Bouman, Denise Tönissen, Judith Stoef

Recoverable robustness

- Method to cope with uncertainty
- Combination of
  - Robust optimization
  - Two-phase stochastic programming
- Related to Ajustable Robustness
- Standard reference:

  Liebchen, Lübbecke, Möhring, Stiller

  LNCS 5868, 2009
- Much work done within the ARRIVAL project

Example: European Soccer Championship 2016

Scheme of the tournament

1. Group phase.
   Six groups: numbers 1 and 2 advance
2. Knock-out phase:
   1. Eight finals
   2. Quarter finals
   3. Semi finals
   4. Final

You want to support your team!

- You want to buy tickets
- Tickets must be bought months in advance
- The schedule for the group is known
- The schedule for the eight finals etc. depends on the results in the group phase
- Which tickets should you buy?

To make it more clear …

- Group A:
  - France (17)
  - Switzerland (15)
  - Albania (42)
  - Romania (23)
- Likely scenario: France A1
- Possible scenario: France A2
- Other scenarios are extremely unlikely to you

Robust Optimization

- Find the cheapest solution
  - You don’t know yet which scenario applies
  - It must be feasible for all scenarios (possibly excluding the very unlikely ones)
  - Recovery actions afterwards are not possible
- Here: buy tickets for all possible matches for A1 and A2
- So for example Lyon July 6 and Marseille July 7

Reference

- Textbook by Ben-Tal Ghaoui and Nemirovski (2009)
- 15.600 publications in years 2010-2015

2-phase stochastic programming

- Birge and Louveaux (1997)
- Assumption: The probability of each scenario is known
- A solution comprises of first stage and second stage decisions; its cost is the expected cost
- The first stage decisions render a solution that can be made feasible through the second stage decisions when the real scenario gets revealed

For our example:

- First stage: buy only tickets for the group phase
- Second stage: acquire the tickets for the knock-out phase in some way (black market???)

Recoverable robustness

- Robust optimization: too conservative
- 2-Phase stochastic programming: too difficult

- Recoverable robustness:
  - Find an initial solution (first stage decision)
  - **You must be able to make it feasible using a simple recovery algorithm (possibly excluding some very unlikely events)**
  - The cost is a combination of the first and second stage decisions

Recoverable robustness: example

- Exclude very unlikely events: You `know' France will advance
- Buy tickets for all matches of A1 (or A2 if you want to see Switzerland)
- Simple recovery if necessary: exchange tickets with a supporter of the other team (make trade, not war).

Cheap and simple

And now back to the real subject ...

Outline

- Introduction and background
- Size robust knapsack problem
  - Models
  - Decomposition approaches
- Scheduling: minimizing the number of late jobs
- Demand robust shortest path

INTRODUCTION AND BACKGROUND

Recoverable robust optimization: formal definitio

Optimization problem

$$P = \min\{f(x)|x \in F\},$$

- Disturbances: discrete scenarios S.
- F_s: set of feasible solutions for scenario $s \in S$,
- y^s: decision variables for scenario s
- $\mathcal{A}$ set of recovery algorithms, where $A(x, s) \in \mathcal{A}$ determines a feasible solution y^s from a given initial solution x in case of scenario s.
- $c^s(y^s)$: the cost of recovery variables y^s,

The *recovery robust* optimization problem is now defined as:

$$\mathcal{RRP}_{\mathcal{A}} = \min\{f(x) + g(\{c^s(y^s)|s \in S\})|x \in F,\ A \in \mathcal{A},\ \forall_{s \in S}\, y^s = A(x, s)\}.$$

Recoverable robust optimization

The function *g* combines the cost of the recovery variables into the objective function.

Possibilities for *g:*

- *All-zero:* we just require feasibility
- *Maximum:* we minimize the worst-case
- Suppose we know probabilities p_s for each of the scenarios *s*, then we can use *expected cost* $\sum_{s \in S} p_s c^s(y^s)$.

Adjustable robust optimization

- Related concept
- Introduced in Ben-Tal et al. (2004)
- Multi-period decision problem, 2 or more periods.
- Optimization of worst case
- Uncertainty set Z. Time 2 decision vector x_2 is a function of realised scenario $\zeta \in Z$.

Column Generation

- Technique based on Linear Programming
- Typical Problem:
  - We are looking for a *good combination* of partial, separated solutions, subject to some constraints (covering problems)
  - The number of partial solutions is very large, while our final solution will combine only a few
- Decomposition approach
  - Introduce a Master Problem for combining partial solutions
  - Derive dual values (shadow prices) from the MP
  - Derive Pricing Problems using these duals to find partial solutions that will improve the solution to the MP

Column Generation

- Examples from research at UU CS department
  - Gate assignment at Schiphol Airport
  - Scheduling in public transport (trains, buses, ...)
  - Scheduling of classes in high school or universities
  - The list goes on…
- Example in this talk: Shawarma (©Paul Bouman)
  - We need 3 ingredients: meat, bread and sauce
  - For 200 people, so we want a lot
  - We consider package deals, but don't know all possible deals (bargaining is possible)
  - We want to find the cheapest combination of package deals

Column Generation

Minimize

dealer				
current amount				
price				
…	≥	$50kg$		…
…	≥	400		…
…	≥	$20l$		…

Total: € …

Column Generation

Minimize

					
40	10				
€8,00	€5,00				
$1{,}5kg$	$0.75kg$	$\geq$	$50kg$		0
8	8	$\geq$	400		0,625
$0{,}5l$	$0l$	$\geq$	$20l$		6

Total: € 370

Column Generation

Minimize

Can we find a new deal with price c such that, if a_b is the amount of bread and a_s is the amount of sauce, we have $c < 0{,}625\ a_b + 6\ a_s$?

I can sell you packages of 2 breads and 0,5 l sauce for €4 (And $0{,}625 \cdot 2 + 6 \cdot 0{,}5 > 4$ is valid)

OK !

Column Generation

Minimize

						
$18\ ^{2}/_{3}$	$21\ ^{1}/_{3}$	24				
€4,00	€8,00	€5,00				
0	$1{,}5kg$	$0.75kg$	≥	$50kg$		0,266
2	8	8	≥	400		0,6
$0{,}5l$	$0{,}5l$	$0l$	≥	$20l$		5,6

Total: € 365

Minimize

						
$18\ ^{2}/_{3}$						
€4,00	€8,00	€5,00				
		$.75kg$	$\geq$	$50kg$		0,266
2	8			00		,6
$0{,}5l$	$0{,}5l$					5,6

Can we find a new deal with costs c, amount of meat a_m, amount of bread a_b and amount of sauce a_s, such that $c < 0{,}266\ a_m + 0{,}6\ a_b + 5{,}6\ a_s$?

I can sell you packages of 0,25 kg of meat and 7 breads for € 3 (and $0{,}266 \cdot 0{,}25 + 0{,}6 \cdot 7 > 3$ is valid)

OK !

Column Generation

Minimize

								
20	10	30	0					
€3,00	€4,00	€8,00	€5,00					
$0{,}25kg$	0	$1{,}5kg$	$0.75kg$	≥	$50kg$		1,11
7	2	8	8	≥	400		0,38
0	$0{,}5l$	$0{,}5l$	$0l$	≥	$20l$		6,44

Total: € 340

Column generation for LP

1. Start with Restricted Master Problem: a small set of partial solution (deals)
2. Solve LP-relaxation.
3. Find out if there is a new partial solutions (deals) that can improve the solution=

   Pricing: Find partial solution (deal) with minimal reduced cost

   - If minimum < 0, add column to model and go to 2.
   - Otherwise stop ⇒ optimum found

Description based on minimization (for maximization we maximize the reduced cost and add column with positive reduced cost)

SIZE ROBUST KNAPSACK PROBLEM

Size Robust Knapsack Problems

- Example: you must select projects, but the budget may become lower than originally anticipated.
- Standard knapsack items with deterministic weight and value
- The volume of the knapsack is *b*, but this value may be smaller than originally assumed
- Set of scenarios: scenario s has volume b_s with probability p_s (scenario 0 is the original case)

Size Robust Knapsack (2)

- Possible objectives:
  - Optimize for initial situation and require feasibility in scenarios
  - Worst-case profit
  - Expected profit
- Some possible recovery rules
  - Recovery by removal
  - Greedy recovery (remove items with largest weight, remove items with minimal value, or minimal value per unit weight)
  - Recovery by swapping
  - Recovery can be cardinality constrained

Size Robust Knapsack (3)

- Dynamic programming for different variants
- Then focus on expected profit and recovery by removal
  - Decomposition framework
  - Other algorithms

Size Robust Knapsack Problems: Dynamic programming

Objective	Recovery	Runtime
Feasibility	Removal, Cardinality constrained ($\leq k$)	$O(nbk)$
Expected profit	Greedy Recovery	$O(nb\|S\| + n \log n)$
Expected profit	Swapping	$O(nb(n+\|S\|))$
Expected profit	Removal	$O(nb^{\|S\|})$

Expected profit, greedy recovery

- Order the items on recovery criterion, e.g increasing weight,
- Hence you will always remove the item with largest index.

- State variable: *E(i,w)*: best solution value using items from *{1,…,i}* and knapsack weight *w*.

- $E(i,w) = \max\{E(i-1,w), E(i-1,w-a_i) + \sum_{s\in S:b_s\geq w} p_s c_i\}$

Expected profit and swapping

- Use state variable $D(i,k,w)$ the best value of a knapsack with item from $\{,1,2,\cdots,i\}$ cardinality k and weight w.
- $D(i,k.w) = \max\{D(i-1,k,w), D(i-1,k-1,w-a_i)+c_i\}$
- Observation: if we do recovery by swapping, then the knapsack has the same cardinality in each scenario.
- Algorithm:
  - Compute the table of $D(i,k,w)$
  - For each k, determine the total expected value if we take for each scenario the best feasible knapsack of cardinality k
  - The optimum is the maximum over k

Expected profit and recovery by removal

2 scenarios

- State variable $F_2(i, w_0, w_1)$ best value of a knapsack with item from $\{,1,2,\cdots,i\}$ where the intial knapsak has weight w_0 and the recovery knapsack has weight w_1

- $F_2(i, w_0, w_1) =$
$$\max\{F_2(i-1, w_0, w_1), F_2(i-1, w_0 - a_i, w_1) + p_0 c_i,$$
$$F_2(i, w_0 - a_i, w_1 - a_i) + (p_0 + p_1)c_i\}$$
- Generalize to $F_{|S|}(i, w_0, w_1, w_2, \cdots w_{|S|-1})$
- Complexity $O(nb^{|S|})$.

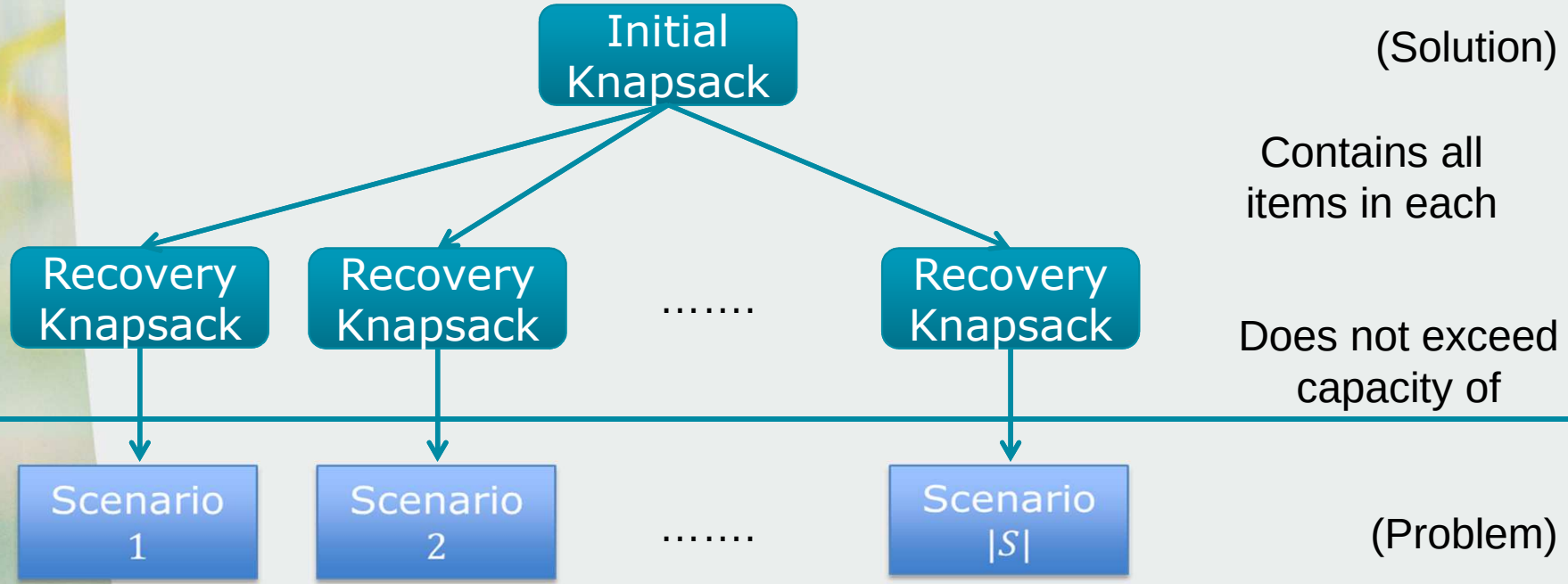
Scales poorly in the number of scenarios!
(Is this the best DP we can think off?)

Column generation framework

- Two variants:
  - Separate recovery decomposition
  - Combined recovery decomposition
- This framework can be applied to many problems
  - Size robust knapsack
  - Maximize number of on-time jobs
  - Shortest path with unknown destination
  - Maximum weighted independent set with an expanding graph
  - Max flow with decreasing capacities
  - …
- Now explained for size robust knapsack with objective expected profit and recovery by removal

Decomposition Framework

- Lets consider this structure for our size-robust Knapsack Problem with recovery by removal



Separate recovery decomposition

What we want:

1. A single initial solution
2. A single solution for each scenario
3. Items selected in the scenario solution must form a subset of the items selected in the original solution (recovery by removal)

ILP-formulation

Given all possible knapsack fillings, use indicator variables + constraints to enforce (1)-(3).

Use branch-and-price to solve it.

Separate Recovery Decomposition: Master problem formulation

- $K(b)$ set of feasible knapsack fillings of size b
- Decision variables $x_k = \begin{cases} 1, & if\ k\ \in K(b) is\ selected \\ 0, & otherwise \end{cases}$
- Decision variables $y_q^s = \begin{cases} 1, & if\ k\ \in K(b_q) is\ selected \\ 0, & otherwise \end{cases}$
- Choose a single initial solution
- Choose a single recovery solution for each scenario $s \in S$
- Linear recovery constraints demand that each recovery solution is recoverable from the initial solution

$$\max p_0 \sum_{k \in K(b)} C_k x_k + \sum_{s \in S} p_s \sum_{q \in K(b_s)} C_q^s y_q^s$$

subject to

$$\sum_{k \in K(b)} x_k = 1$$

$$\sum_{q \in K(b_s)} y_q^s = 1 \quad \text{for all } s \in S$$

$$\sum_{k \in K(b)} a_{ik} x_k - \sum_{q \in K(b_s)} a_{iq}^s y_q^s \geq 0 \quad \text{for all } i \in \{1, 2, \ldots, n\}, s \in S$$

$$x_k \in \{0, 1\} \quad \text{for all } k \in K(b)$$

$$y_q^s \in \{0, 1\} \quad \text{for all } q \in K(b_s), s \in S,$$

where the parameters a_{ik} and a_{iq}^s are defined as follows:

$$a_{ik} = \begin{cases} 1 & \text{if item } i \text{ is in knapsack } k \in K(b), \\ 0 & \text{otherwise.} \end{cases}$$

and

$$a_{iq}^s = \begin{cases} 1 & \text{if item } i \text{ is in knapsack } q \in K(b_s), \\ 0 & \text{otherwise.} \end{cases}$$

Duals

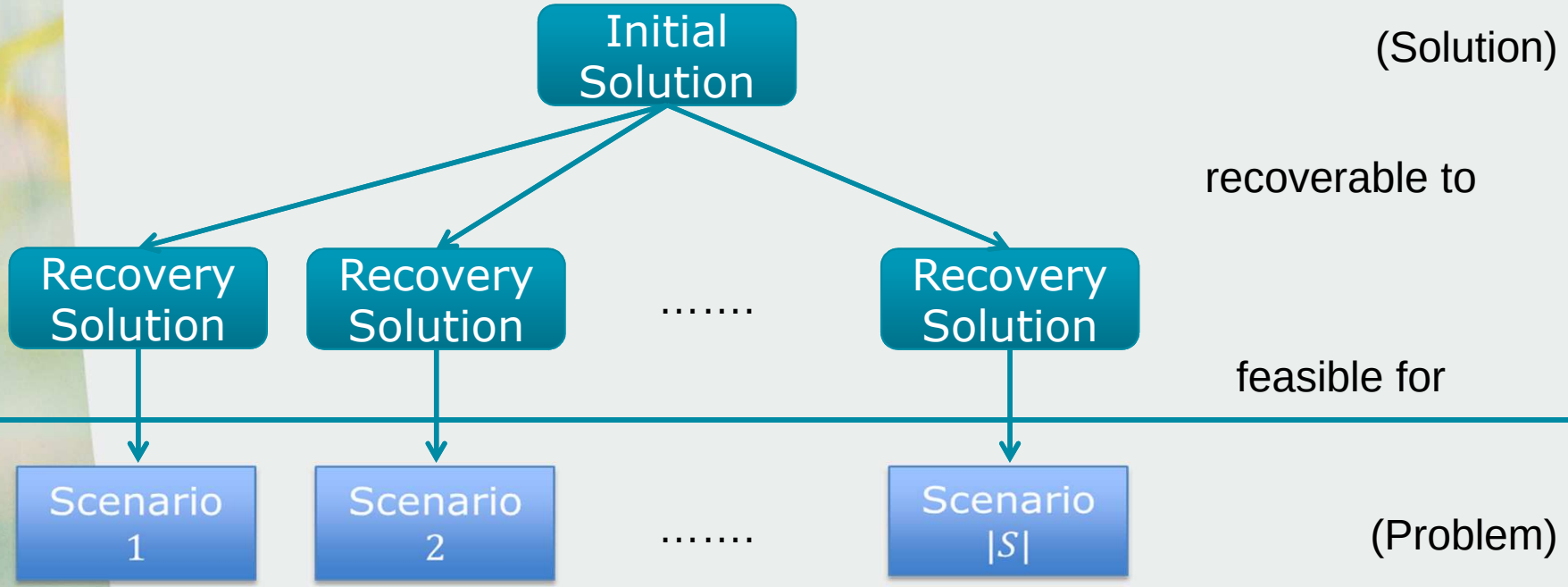
λ

μ_s

π_{is}

Separate recovery decomposition

- The Separate Recovery decomposition considers initial solutions and recovery solutions **separately**



Column generation for LP (maximization)

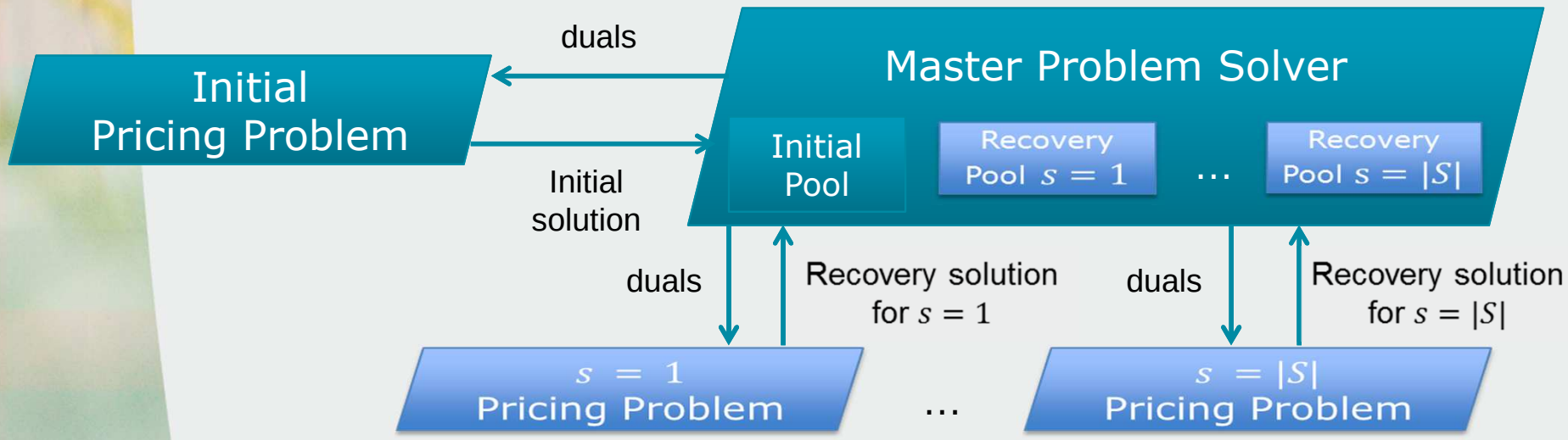
1. Start with Restricted Master Problem: a small set of partial solutions
2. Solve LP-relaxation.
3. Find out if there is a new partial solutions that can improve the solution=

   Pricing: Find partial solution (deal) with maximal reduced cost

   - If minimum > 0, add column to model and go to 2.
   - Otherwise stop ⇒ optimum found

Separate Recovery Decomposition

- Components in the implementation



Separate Recovery Decomposition: Pricing Problem initial solution

Reduced cost

$$
\begin{aligned}
c^{\text{red}}(x_k) &= p_0 \sum_{i \in k} c_i - \lambda - \sum_{i=1}^{n} \sum_{s \in S} a_{ik} \pi_{is} \\
&= \sum_{i=1}^{n} a_{ik} (p_0 c_i - \sum_{s \in S} \pi_{is}) - \lambda.
\end{aligned}
$$

hence we have to find column *k* for which reduced cost is maximal

Column *k* represented by feasible $(1, 0, \ldots 0, a_{1k}, a_{2k} \cdots a_{nk})$

Knapsack problem!

Separate Recovery Decomposition: Pricing Problem recovery solution

- Reduced cost

$$c^{\text{red}}(y_q^s) = \sum_{i=1}^{n} a_{iq}^s (p_s c_i + \pi_{is}) - \mu_s$$

For each scenario *s*
we have to find column *q* for which reduced cost is maximal
Column *q* represented by feasible $(0, \ldots 0,1,0, \ldots 0, -a_{1q}^s \quad \cdots -a_{nq}^s)$

- **Knapsack problem for knapsack with size b_s!**

Combined recovery decomposition

Alternative approach:

Consider combinations of an initial solution with a scenario solution

- Enumerate all possible combinations per scenario
- Select one combination per scenario
- See to it that the initial solution is the same in each selected combination.

Use branch-and-price to solve this ILP.

Knapsack – Combined Recovery Decomposition

- Constants
  - C_j : The profit of knapsack j
  - a_{ik} : Is 1 if item i is in knapsack k, 0 otherwise
- Binary Variables
  - x_i : Item i is selected in the initial solution
  - $y^s_{(k,j)}$: k is selected as an initial knapsack, j as a recovery knapsack for scenario s
- Master Problem:
  - Maximize $\sum_i p_0 c_i x_i + \sum_{s \in S} \sum_{(k,j)} p_s C_j y^s_{(k,j)}$
  - Subject to
    - $\sum_{(k,j):feasible_s(k,j)} y^s_{(k,j)} = 1 \quad \forall s \in$
    - $x_i = \sum_{(k,j)} a_{ik} y^s_{(k,j)} \quad \forall i \in I, \forall s \in S$

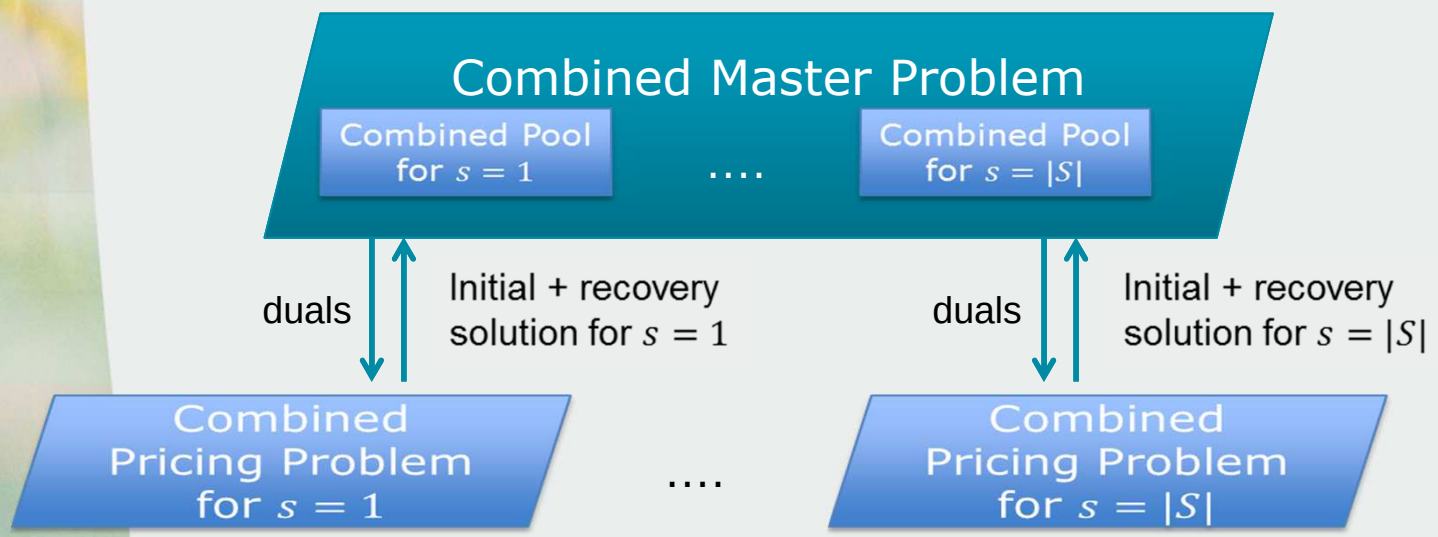
An item is selected in the initial solution, if and only if it is selected in the initial solution of any scenario

Combined Recovery Decomposition

- Master Problem
  - Choose a single initial/recovery combination for each scenario $s \in S$
  - The initial decision variables in our master problem should be equal to the value of that decision variable in the selected initial solution for each scenario $s \in S$
- Pricing Problem for each scenario $s \in S$
  - Select a feasible combination of an initial and a recovery solution with the best reduced costs

Combined Recovery Decomposition

- Components in the implementation



Computational experiments

- Separate recovery decomposition: branch-and-price
- Combined recovery decomposition: branch-and-price
- Branch-and-bound:
  - Upper bound from standard LP-formulation
  - Lower boud based on DP for initial knapsack
  - Branching on inclusion of items in initial knapsack
- Dynamic programming
- Hill-climbing
  - Search on initial solutions
  - Optimal recovery applied for each initial solutions

- Instances up to 100 items based on classes of difficult knapsack instances from Pisinger (2005)

Computational experiments

- Dynamic programming performed worst
- Separate recovery outperformed branch-and-bound by completing more instances within the time limit.
- Separate decomposition: very quick, solves problems with 100 items and 20 scenarios in seconds, but it can be slow on bad instances. Increasing the number of scenarios does not increase the running times.
- Combined decomposition: much slower. Presumably due to the more difficult pricing problem (quadratic in b).
- Hill-climbing: quick but might find only 85% of optimal value

MINIMIZE NUMBER OF TARDY JOBS

Minimize number of tardy jobs

- 1 machine
- n jobs become available at time 0
- Known processing time p_j
- Known due date d_j
- Known reward w_j for timely completion (currently 1)

Decision to make at time 0: **accept or reject** job j

Moore-Hodgson (MH)

1. Number the jobs in EDD order
2. Let S denote the EDD schedule
3. Find the first job not on time in S (suppose this is job j)
4. Remove from S the largest available job from jobs 1,…,j
5. Continue with Step 3 for this new schedule S until all jobs are on time

Stochastic processing times

- If the jobs follow some nice distribution, then use **chance constraints:**

  Job j is on time if $P(C_j \leq d_j) \geq y_j$

  for a given threshold y_j
- The problem can then be solved by adjusting the due dates and processing times through Moore – Hodgson (vdA, H)

Disturbed processing times

- Different situation: processing times do not follow some nice probability distribution, but may get disturbed by some event
- We model these disturbances as **scenarios**
- Each scenario has a given probability
- In each scenario the processing times are given, deterministic values.

Situation under consideration

- The basic processing times and all possible scenarios are known
- At time zero you must decide which jobs to accept/reject
- Then the actual scenario gets revealed

- *How to deal with these disturbances, i.e. which jobs are removed?*

- Maximize then expected number of on-time job (i.e. minimize the expected number of late jobs)

Observations

- Given the initial solution, you can solve each scenario to optimality by applying MH to the set of accepted jobs
- Applying MH to find the initial solution is not always optimal (not even if there is only one scenario)

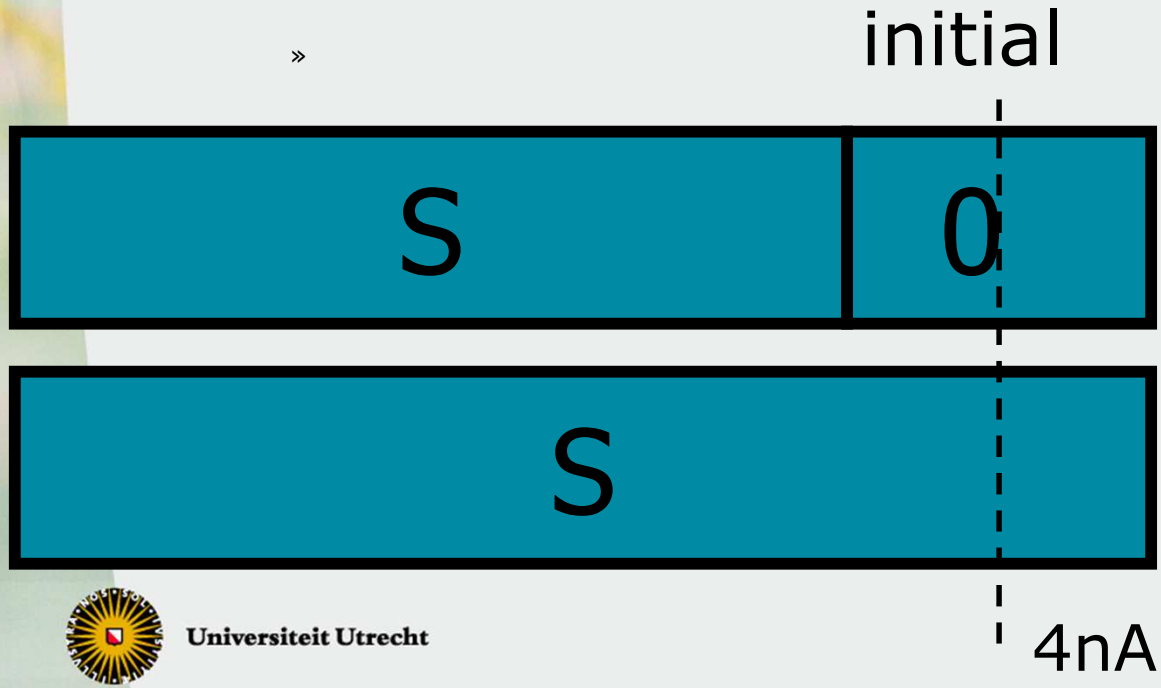
Computational complexity

- The problem is NP-hard in the ordinary sense, even in case of one scenario (which has probability *p)* and a common due date

- Reduction from Partition with *equal cardinality*: 2n nonnegative integers a_j with total sum 2A. Does there exist a subset *S* of cardinality *n* such that $\sum_j a_j = \mathrm{A}$.

Some details

- Each a_j leads to a job j with $p_j = 4A - a_j$ and $p'_j = 4A + a_j$; job 0 has $p_0 = 2A$ and $p'_0 = 40A$.
- Common due date D=4nA+A.
- Yes in partition iff expected number of on-time jobs= $n + (1 - p)$ in scheduling problem



Further investigations

- Apply MH both to the initial instance and the scenario instance.
- If job j gets selected in both, will it be in any optimal solution?
- If job j gets selected in none, can you discard it without losing optimality?
- Answer is NO in both cases!

Dominance rule

- Define H' as the jobs that are on time when MH is applied to the initial instance and the scenarios (there can be more than 1 scenario).
- Define H as the subset of H' that contains all jobs j with
  - $p_j \leq p_i$ for all jobs i outside H for
  - $P_{js} \leq p_{is}$ for all jobs i outside H and for all scenarios s

Dominance rule

- There exists an optimal solution in which all jobs from H are selected in the initial solution (and hence in each scenario solution)
- If processing time are non-decreasing in the scenarios, then there exists an optimal solution in which all jobs from H’ are selected in the initial solution.

Solution algorithms: B&B

- Remark: we only need to know the best choice for the initial instance (scenario solution follows by applying MH)
- Branch on whether job j is on time or late in the initial solution
- Lower bound: Solve the initial and scenario instances independently

Dynamic programming

- Add jobs in earliest-due-date order
- Use state-variables $f_j(t_0, t_1, \ldots, t_k)$, where t_j indicates the total length of the accepted jobs for scenario j (scenario 0 is initial).
- Running time becomes problematic if the number of scenarios becomes big

Branch-and-price

- Apply Separate Recovery Decomposition
  - Use binary variables to decide whether a schedule is chosen as initial or scenario solution
  - Use constraints to enforce that the chosen initial and scenario solutions are compatible.
  - The LP-relaxation is solved using column generation.
  - New schedules are selected by solving the pricing problem, which can be solved efficiently using DP.

Experiments

- Basic ILP (not shown here) performs worst
- DP becomes very slow in case of many scenarios
- Branch-and-bound performs quite well; current implementation can be improved
- Branch-and-price does well if the LP-relaxation is integral (20% of the instances with 120 jobs and 10 scenarios). It is much slower otherwise, but comparable to B&B.

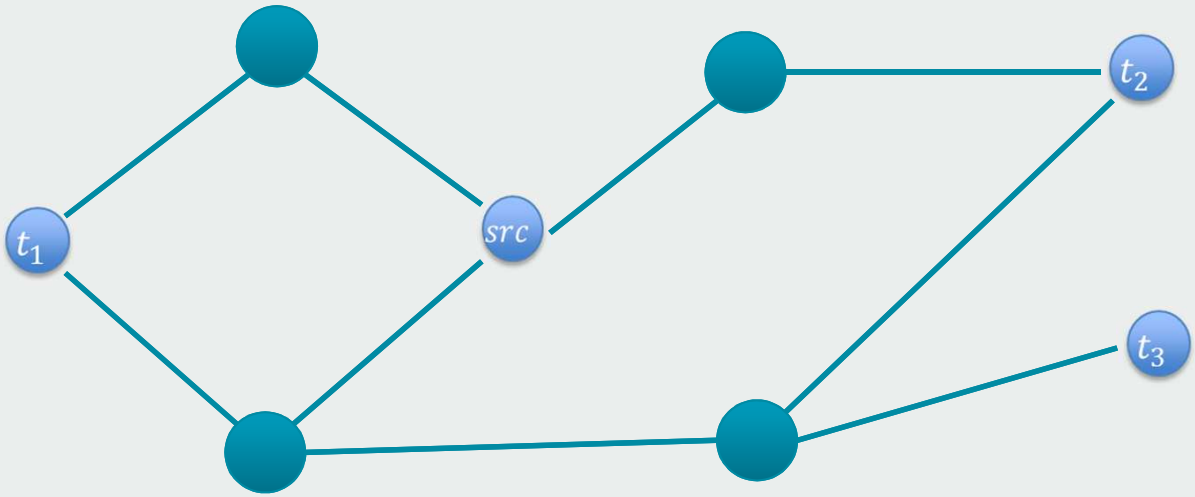
Column generation framework

- Two variants:
  - Separate recovery decomposition
  - Combined recovery decomposition
- This framework can be applied to many problems
  - Size robust knapsack
  - Shortest path with unknown destination
  - Maximum weighted independent set with an expanding graph
  - Max flow with decreasing capacities
  - …

DEMAND ROBUST SHORTEST PATH

Demand Robust Shortest Path

- Shortest path problem
- Single source *scr*, scenarios define sinks
- Buy edges during a cheaper initial phase
- When sink is known, edges are more expensive



Demand Robust Shortest Path

- Separate recovery decomposition: hard to express the connection between the original solution and the scenario (original edge set + scenario edge set = path)
- Combined Recovery Decomposition?

Demand Robust Shortest Path: Combined Recovery Decomposition

- find an original + scenario edge set that is a path : simple problem. Use it!
- Pricing problem becomes:
  - Two options for each edge – just take the cheapest
  - Only initial prices can be negative due to duals. We take these no matter what and consider their costs to be 0
  - Find the shortest path in a graph with non-negative edge weights

Solve using Dijkstra's Shortest Path Algorithm!

Column generation framework

- Two variants:
  - Separate recovery decomposition
  - Combined recovery decomposition
- Theoretical general result: the LP-relaxation of Combined Recovery Decomposition is stronger
- The computation time of Combinatorial Recovery Decomposition can be reduced by clever column generation strategies

Conclusions

- The type of recovery has a lot of impact on finding an algorithm to the problem
- The column generation framework is a great way to reduce recoverable robustness problems into regular problems (while customization remains possible)
- We believe that it can be applied to many different problems