Section 3.5:

Backtracking Strategies

Decision strategies

- Like in the greedy algorithm, we will descend along individual paths of the decision tree (more than one path now).
- For that, we need a strategy which option to choose at every node.
- Roughly speaking, the decision strategy in the greedy algorithm was:

  Always choose the option that looks best possible (without any foresight).

- However, this is but one possible decision strategy.

  $\longrightarrow$ See another example on the next slide.

Example: feature-based problems on independence systems

- Again, the following is but an example of decision strategies for feature-based problems on independence systems.

  $\longrightarrow$ Probably better than greedy but usually not good.

- As usual, the root represents an empty selection of features.
- *Decision* at a node v:
  - ◇ One option for every feature that can be added to the selection of features corresponding to v, provided it does not make this selection infeasible.
  - ◇ From each child node of v (=option at decision v), a greedy path down to a leaf (=solution) is computed.
  - ◇ The option for which the best solution was found by that is chosen for the next step down.

Generic scheme of backtracking

- Go down according to a certain *default* decision strategy until
  - ◇ either a feasible solution is found
  - ◇ or the search gets stuck at a node where it is obvious that no feasible solution can be reached.
- If a solution is found: terminate.
- If some additional termination criterion (e.g. limited CPU time) applies: terminate.
- Otherwise, analyze the problem, change the decision strategy accordingly, and repeat the entire procedure.

  $\longrightarrow$ See the next slide.

Default decision strategy: often (not exclusively), a strategy of the greedy type.

"... analyze the problem, change the ... strategy, repeat ..."

- If the default decision strategy is a greedy strategy, then, by definition, every option chosen on the way from the root to the leaf in the decision tree "looks good" at the time when that decision is made.
- However, some of these decisions may be responsible for a bad final result.
- The idea is
  - ◇ to identify these decisions,
  - ◇ to disallow the options chosen for them in the greedy algorithm, and
  - ◇ to start the greedy algorithm again, but obeying these exclusions.

Example: machine scheduling

- The greedy algorithm from Section 2.3.4 is a reasonable default decision strategy (not the only one, of course).
- After applying the greedy algorithm, we try to find another solution with a smaller completion time.
- General backtracking strategy:
  - ◇ In the greedy solution, look for decisions that seem to be responsible for an unnecessarily high completion time.

    $\longrightarrow$ "Trouble makers"
  - ◇ Disallow the chosen options for these decisions.

Remove "trouble makers"

- Simple strategy:
  - ◇ Select some of the items on the "bottleneck" machines (preferably items with long durations).
  - ◇ Disallow their assignments to this machine.
- Smarter strategy:
  - ◇ Look at the the predecessors of these items with respect to the precedence constraints.
    $\longrightarrow$ And the predecessors of the predecessors, etc.
  - ◇ Identify items that enforce other items to be placed on bottleneck machines.
  - ◇ Disallow their assignments.
- Even smarter strategies are imaginable.

Iterated procedure

- As said before, this backtracking is repeated time and again.
- It is not advisable to let all exclusions of options valid because
  - ◇ keeping all "old" exclusions valid while adding new ones may be unnecessarily overrestrictive (the new exclusions may also resolve the old problems to some extent) and
  - ◇ an ever increasing set of exclusions would, step-by-step, eliminate all flexibility from the procedure.

Backtracking – what's in a name?

- Often, exactly one decision is identified as a "trouble maker" and the option chosen for that decision disallowed.
- We can alternatively implement this variant as follows:
  1. Go down according to the default decision strategy.
  2. While the result is not satisfactory:
     (a) Go back on the chosen path in the decision tree, step-by-step. $\longrightarrow$ Backtracking
     (b) Stop when a "trouble maker" is found.
     (c) Choose another way down at that node (=option for that decision).
     (d) From there, apply the default decision strategy again to go down up to another leaf.

Section 3.6:

Cutting Strategies

Cut-offs

- In Section 3.4 we have seen that, in rare cases, we can afford to explore the decision tree exhaustively.
- In most cases, the decision tree is way too large, so we can only afford to explore parts of it.
- In other words, whenever we visit a node of the decision tree, we will make a decision
  - ◇ whether to descend into the subtree rooted at this node
  - ◇ or to discard this subtree as a whole.
- In Sections 3.6.1–3.6.3, we will consider three different techniques that cut off subtrees only if these subtrees are definitely useless.

Definitely useless subtrees

A subtree of the decision tree is definitely useless...

- in case of a pure feasibility problem:

  if the instance is infeasible **or** there is at least one *feasible* solution outside the union of this subtree and all subtrees cut off previously.

- in case of an optimization problem:

  if the instance is infeasible or unbounded **or** there is at least one *optimal* solution outside the union of this subtree and all subtrees cut off previously.

Determining uselessness

- To cut off a subtree without losing accuracy, we will compute some kind of evidence that this subtree is definitely useless.
- The techniques in Sections 3.6.1–3.6.3 will basically differ by the very nature of the computation strategy.
- *Side remark*: we will see that different traversal strategies are appropriate for the individual types of computation strategies (cf. Section 3.3).
- *Note*: for $\mathcal{NP}$-hard algorithmic problems, an efficient strategy cannot perfectly accurately determine whether a subtree is definitely useless or not.

  $\longrightarrow$ Otherwise, applying this strategy to the root of the decision tree would efficiently determine whether the instance is feasible.

Conservative determination of uselessness

- So this means we cannot get perfectly accurate evidence within a reasonable amount of run time.
- Alternatively, we will aim at a *conservative* strategy.
- This means two outcomes are possible: "yes, definitely useless" or "don't know".
- It goes without saying that the subtree must indeed be definitely useless whenever "yes, definitely useless" is the answer.

  $\longrightarrow$ We are on the "safe side" = conservative strategy.
- The challenge is to design strategies such that "don't know" is not too often the outcome in cases where the subtree is indeed definitely useless.

Remark on efficieny

- In practice, even a good strategy may leave too many nodes of the decision tree to be explored.
- This is at least the case whenever too many nodes of the decision tree have feasible instances as leaves of their sub-trees.
- In such a case, there are several options.

  $\longrightarrow$ See the next slide.

Options

- Terminate the exploration earlier and prematurely.

  $\longrightarrow$ Cut off all unexplored parts of the tree, useless or not.

- Use a cut-off strategy that is not perfectly conservative.

  $\longrightarrow$ Useful subtrees may be cut off.

- Switch to another algorithmic strategy to explore selected subtrees.

  $\longrightarrow$ Local-search based techniques may be promising in such a case.

Section 3.6.1: Constraint Programming

What is constraint programming?

- Roughly speaking, this technique tries to certify for a given subtree of the decision tree that none of its leaves is a feasible solution.
- From Section 3.1, recall that the nodes of the decision tree may be identified with subsets of the solution space.
- For a better understanding, we will interpret the edges of the decision tree as additional constraints.
- *Example* feature-based problems:
  - ◇ The edges leaving a node of the decision tree on level $i-1$ represent the options whether or not to select feature i: $x_i = 1$ or $x_i = 0$.
  - ◇ Thus, one edge leaving this node represents the additional constraint $x_i = 1$, the other edge represents the additional constraint $x_i = 0$.

Determining uselessness of subtrees

- So, a node v of the decision tree may be identified with the set of additional constraints on the branch from the root to v.
- The original constraints of the instance plus these additional constraints may make all leaves of the subtree rooted at v infeasible.
- Constraint programming aims at detecting this situation as often and as early as possible.
- As we said in the introduction to Section 3.6, not all cases can be detected with absolute certainty.

Illustrative "toy" example: n-queens problem

- Consider an $(n \times n)$-chessboard.
- When a queen is placed on a square, it may threaten all squares in the same column and row and in all four diagonal directions (like in the chess game).
- The problem is to place as many queens as possible on different squares of this chessboard such that no two queens threaten each other.
- Of course, the maximal number of queens is bounded from above by n.

Example n-queens problem (cont'd)

- Idea for a decision tree (of height n):
  - ◇ A node on level i represents the decision where to place the i-th queen (if still possible).
  - ◇ As many as n^2 arcs leave a node: one for each position.
- So, every node of the decision tree on height h represents a particular placement of as many as h queens.
- Whenever a node is visited, it is first checked whether there is still a position left for yet another queen.
- If not, this subtree is cut off (what else?).

Constraint propagation

- In the last example, it was easy to determine at each node of the decision tree whether we may cut off the subtree rooted at this node.
- However, often we need sophisticated algorithms for that.
- Typically, such an algorithm combines the existing constraints to generate new, redundant constraints.

  $\longrightarrow$ The original constraints of the instance plus all constraints corresponding to the edges from the root to the current node.
- The heuristic hope is that eventually a set of constraints is generated from which infeasibility can be immediately concluded.

Very simple and (hopefully) instructive, yet artificial example

Find x and y subject to

$x \in \{1, 2, ..., 9\}$ AND
$y \in \{1, 2, ..., 9\}$ AND
$x + y = 9$ AND
$2x + 4y = 24$ AND
$x < 2y$

Constraint propagation steps in the example

- $x \in \{1, 2, ..., 9\}$ AND $y \in \{1, 2, ..., 9\}$ AND $2x + 4y = 24$

  $\longrightarrow$ $y \geq 2$ AND $x \leq 8$ AND $y \leq 6$
- $x \in \{1, 2, ..., 8\}$ AND $y \in \{2, 3, ..., 6\}$ AND $x + y = 9$

  $\longrightarrow$ $x \geq 3$ AND $x \leq 7$
- $x \in \{3, 4, ..., 7\}$ AND $y \in \{2, 3, ..., 6\}$ AND $2x + 4y = 24$

  $\longrightarrow$ $x \geq 4$ AND $x \leq 6$ AND $y \geq 3$ AND $y \leq 4$
- $x \in \{4, 5, 6\}$ AND $y \in \{3, 4\}$ AND $x + y = 9$

  $\longrightarrow$ $x \geq 5$
- $x \in \{5, 6\}$ AND $y \in \{3, 4\}$ AND $2x + 4y = 24$

  $\longrightarrow$ $x = 6$ AND $y = 3$
- $x = 6$ AND $y = 3$ AND $x < 2y$

  $\longrightarrow$ Infeasible!

More realistic example: machine scheduling

- From Section 2.3.4, recall the machine scheduling problem.
- Basically, we got N items with durations $d_1, \ldots, d_N$ and precedence constraints among some pairs of items.

  $\longrightarrow$ Limits the possibilities to process several items simultaneously.
- We may also have constraints due to restricted resources.

  $\longrightarrow$ Limits the possibilities to process several items simultaneously even further.
- For the individual items, we also assume earliest and latest permitted start times.

  $\longrightarrow$ A "time window" for each item.

Specific example – house construction

- The items are individual tasks such as building the basement, building a wall, inserting a window, etc.

  $\longrightarrow$ An appropriate granularity is to be chosen by the project management.

- Example of precedence constraints: roof construction may only be started after all wall constructions have been finished.
- Examples of restricted resources: cranes, vehicles, handymen, specialists.
- Earliest and latest permitted start times may result from seasons, time-bound contracts with third-party companies, temporary availability of machines, etc.

  $\longrightarrow$ In rare cases, this may result in more than one period [earliest...latest] permitted start time, but we will ignore this complication here for simplicity.

Observation

If there are no restricted resources, the existence of a feasible schedule can be determined by solving a longest path problem.

$\longrightarrow$ Transformation on the next slide.

Reduction to a longest path problem:

- For each item $i \in \{1, \ldots, N\}$ there are nodes v_i and w_i and an arc (v_i, w_i) with length d_i in the graph.
- For each precedence constraint (i, j), there is an arc (w_i, v_j) with length 0.
- Insert a new node r and for each item $i \in \{1, \ldots, N\}$ an arc (r, v_i), whose length is the earliest start time of v_i.
- Determine the longest distances from r to all nodes.

  $\longrightarrow$ For $i \in \{1, \ldots, N\}$, the longest distance from r to v_i is the earliest possible start time with respect to precedence constraints and earliest permitted start times.
- There is a feasible schedule if, and only if, the distance of no v_i is larger than the latest permitted start time of item i.

Idea for an appropriate decision tree in this example

- The edges represent increased earliest start times of selected items = additional, tighter earliest start times constraints.
- Thus, a node v of the decision tree represents the feasible solutions that fulfill
  - ◇ all original constraints and
  - ◇ the higher earliest start times corresponding to the edges on the path from the root of the decision tree to v.
- The idea is to make, step-by-step, the restricted resources redundant by these increased earliest start times.

  $\longrightarrow$ Eventually, resource restrictions will have no effect anymore, so the longest path solution will do.

Idea for how to determine uselessness

- To determine whether the subtree rooted at v is useless, we ignore all restricted resources and reduce the problem to a longest path problem only with respect to
  - ◇ the original earliest permitted start times,
  - ◇ the original precedence constraints, and
  - ◇ the increased earliest start times of all edges on the path from the root of the decision tree to v.
- If the result violates at least one latest permitted start time, there is no feasible solution satisfying all of these constraints simultaneously.

  $\longrightarrow$ The subtree rooted at v is useless and may be cut off.

Idea for appropriately increased earliest start times

- As usual, the additional constraints will be chosen such that the individual options (=edges) at a node cover the subset of feasible solutions corresponding to this node (these options need not be disjoint).
- As said before, we want to make the resource restriction redundant by that.
- For that, we look at the solution to the longest path instance corresponding to this node (see last slide).
- If no restricted resource is overused at any time in this solution, we have found a schedule that satisfies *all* constraints.

  $\longrightarrow$ We may terminate the entire search.
- So, on the next slide, we will consider the case that at least one restricted resource is overused at some moment in time.

Appropriately increased earliest start times (cont'd)

- If at least one restricted resource is overused at some moment in time, we have to increase some earliest start times in order to remove some overuse in a controlled fashion.
- For that, we look at the very earliest moment in time where one of the resources is overused.
- Let $i_1, \ldots, i_k$ denote the items processed by that resource at that moment in time.
- For every $j \in \{1, \ldots, k\}$, there will be one option: increase the earliest start time up to the smallest completion time of all items $i_1, \ldots, i_{j-1}, i_{j+1}, \ldots, i_k$.

Analysis of this strategy

- To resolve this overuse situation, we have to change the start time of at least one of the items $i_1, \ldots, i_k$ to such an extent that this item does not contribute to this overuse anymore.
- By construction of the initial schedule (reduction to a longest path problem), the start times cannot be decreased without violating the earliest start times or precedence constraints.
- Thus, the start time of at least one item i_j must be increased.
- To relax a bit (or even resolve completely) this overuse situation by moving i_j, the start time of i_j must be at least increased to the earliest completion time of the other jobs involved in the overuse (that is, $i_1, \ldots, i_{j-1}, i_{j+1}, \ldots, i_k$).
- Therefore, all k options together cover all possible cases (not necessarily disjointly).

Section 3.6.2: Branch-and-Bound

What is branch-and-bound?

- Branch-and-bound addresses optimization problems in the first place.
- *Recall*: every instance of an optimization problem comes with a real-valued *objective function*.

  $\longrightarrow$ Assigns sort of a *quality value* (profit, cost, ...) to each feasible solution.
- It is not sufficient to find *some* feasible solution.
- In fact, the challenge is to find a solution whose objective function value is very small/large (if not minimal/maximal).
- In the following, we will focus on *min*imization problems for conciseness.

  $\longrightarrow$ *Max*imization is analogous ("mirror-symmetric").

Fundamental concept of branch-and-bound:

- Suppose we know
  - ◇ either an upper bound U on the optimal cost value
  - ◇ or a feasible solution s (let $c(s)$ denote the cost value of s).
- Further suppose, for every node v of the decision tree, we can compute a lower bound $\ell_v \in \mathbb{R}$ on the objective values of all feasible solutions in the subtree rooted at v.
- This subtree is useless
  - ◇ if $\ell_v > U$, because no optimal solution may be in the subtree rooted at v;
  - ◇ if $\ell_v \geq c(s)$, because s is an optimal solution, too, if the subtree rooted at v contains an optimal solution (in which case we had a perfectly matching lower bound, $\ell_v = c(s)$).

Computing an upper bound

- Clearly, the smaller the value U or $c(s)$ is, the more appropriate this value is for our purpose.
- A *good* upper bound U is often hard to compute.
- However, *some* upper bound U is easy to compute in most cases.

  $\longrightarrow$ For example, in feature–based problems, the total sum of all feature costs is an upper bound U (*very* bad in most cases).

Computing a feasible solution

- For some optimization problems, a feasible solution s with a reasonably small value of $c(s)$ is easy to compute.

  $\longrightarrow$ *Example*: For the TSP, a simple optimizer such as the greedy approach from Section 3.2 produces a feasible solution (typically, quite a good one, in fact).

- For other optimization problems, it is even $\mathcal{NP}$–hard to find a feasible solution.

  $\longrightarrow$ *Counterexample*: Timetabling (cf. Section 1.1).

Updating the upper bound or feasible solution

- Whenever the tree traversal encounters a leaf (=feasible solution) s' with $c(s') < U$ or $c(s') < c(s)$, respectively, it is reasonable to replace U or $c(s)$ by $c(s')$.

  $\longrightarrow$ To increase the chance of determining useless subtrees.

- In particular, if the tree traversal strategy is chosen appropriately, a bad upper bound U or the cost of a bad feasible solution s may soon be replaced by the cost of an even better feasible solution s'.

- So, the standard case is that the lower bound ℓ_v for a node v of the decision tree is compared to the cost value $c(s')$ of quite a good solution s', comparison to an abstract upper bound is a rather exceptional and restricted to the initial search steps.

Branch-and-bound and constraint programming

- Branch-and-bound may be viewed as a specific constraint-programming technique for optimization.

  $\longrightarrow$ Not the only possible way to tackle optimization problems using constraint programming, of course, but fairly standard.

- Constraint programming addressed pure feasibility problems in the first place.
- More specifically, at every node v of the decision tree, we tried to find a contradiction in
  - $\diamond$ the original constraints plus
  - $\diamond$ the constraints corresponding to the edges along the path from the root to v.

Branch-and-bound does fit into this scheme:

- The lower bound ℓ_v is computed on the basis of the original constraints and the additional constraints along the path from the root to v.
- The constraint $\ell_v > U$ or $\ell_v \geq c(s)$, respectively, leads to a contradiction (hopefully): "yes, definitely useless" (otherwise, "don't know").

$\longrightarrow$ The constraint $\ell_v > U$ or $\ell_v \geq c(s)$, respectively, is an example of constraint propagation as defined in Section 3.6.1.

Branch–and–bound and tree traversal:

- From Section 3.3, recall the various options to define tree–traversal orders.
- In principle, each of the canonical orders (and any other order) could work well with a branch–and–bound approach.
- However, under normal circumstances, depth-first search seems to be particularly well suited.

  $\longrightarrow$ See the next four slides for a discussion.

Branch-and-bound and tree traversal (cont'd)

- Recall the idea to update the upper bound whenever a new feasible solution is encountered.

  $\longrightarrow$ Since the leaves are the solutions, we want to see leaves, leaves, leaves, as soon as possible.

- In breadth–first search, the leaves are the very last nodes to be visited.

- In best–first search, this unfavourable effect may also occur approximately.

- In contrast, in depth–first search, we will see leaves frequently from the very beginning of the tree traversal.

Branch-and-bound and tree traversal (cont'd)

- However, this is not the whole story yet.
- The performance of branch-and-bound with depth-first search depends crucially on the order in which the leaves are visited.
- In fact, search starts in a large region of the solution space that is
  - ◇ too bad to be worth exploring but
  - ◇ too good to allow cut-offs at nodes of small heights,

  then the search will spend more time than our universe may provide before visiting a good solution.
- So, the challenge is to design the details of the depth-first search carefully to avoid bad situations like that.

Branch-and-bound and tree traversal (cont'd)

- We only considered the question how to generate new, better upper bounds from *leaves*.
- Not considered in detail here: It is often possible (and promising) to generate new, better upper bounds from *internal nodes* of the decision tree as well.
- If things happen to go all right, this may lead to many early cut-offs even in case of breadth-first search and best-first search.
- In summary: Sometimes, breadth-first search or best-first search may outperform the natural-looking choice, depth-first search.

Branch-and-bound and tree traversal (cont'd)

- Additional advantage of depth-first search over breadth-first search and best-first search: requires less space.
- In fact, breadth-first search requires space in the order of the number of leaves in the worst case (during the very final stages of the search).

  $\longrightarrow$ Way too large for any computer that fits into this universe.
- Best-first search may require the same space as breadth-first search in the worst case.
- In contrast, depth-first search only requires space in the order of the height of the tree.

  $\longrightarrow$ Negligible in the overwhelming number of all applications.

Lower bounds for subtrees:

- Recall the interpretation of a decision-tree node v as a subset of the original set of all feasible solutions.

  $\longrightarrow$ The feasible solutions that obey all constraints corresponding to the tree arcs on the path from the root to v.

- *In other words*: We need a lower bound ℓ_v on the optimum objective value inside this subset.

General idea:

- Some of the side constraints are relaxed or even dropped.

  $\longrightarrow$ The optimal objective value of the relaxed instance is a lower bound on the optimal objective value of the original instance.

- Clearly, such a relaxation to compute lower bounds within a branch–and–bound approach only makes sense if the relaxed problem is significantly easier to solve than the original problem.

Methodological obstacle:

The subset of the solution space corresponding to a node of the decision tree need not be the solution space of some instance of the problem.

$\longrightarrow$ See the next slide for an example.

Example: TSP

- Consider the TSP as a feature-based problem as defined in Section 2.3.2.
- Recall the decision tree for feature-based problems as defined in Section 3.1.
- Let v be a node of this decision tree on height h.

  $\longrightarrow$ For the first h features, a decision has been made whether to select or to reject each of them.
- Let X denote the set of selected features and Y the set of rejected features.
- Then v corresponds to the problem of finding an optimal round tour − among all round tours that cover X and avoid Y.

Discussion

- This is not an instance of the pure TSP anymore!
- But it is an instance of the following more general problem:
  - ◇ *Input*: pairwise distances for N objects, sets X and Y of pairs of these objects (=arcs).
  - ◇ *Output*: a round tour on these N objects such that all pairs from X are contained but no pair from Y.
  - ◇ *Objective*: minimizing the total length of the round tour.

    $\longrightarrow$ Like in the ordinary TSP.

Example I: LP–relaxation

- Consider an optimization problem that is formulated as a special case of ILP.

  $\longrightarrow$ See Examples 7-8 in Section 1.1 for a definition of LP and ILP and for TSP as an example of ILP.

- Dropping the constraint that the solution be integral (and changing nothing else) yields the so–called *LP-relaxation* of this algorithmic problem, which is an LP.
- Since the LP-relaxation is indeed a relaxation, its objective value a lower bound on the objective value of the original ILP.
- In practice, lower bounds from LP–relaxations are often (but not always!) quite tight.
- LP is *much* more efficient to solve than ILP.

  $\longrightarrow$ Not proved here.

In particular, LP-relaxation in the 0/1–case:

- Consider a variable x that is restricted to the values 0 and 1.
- This means three things: x has to be integral, $x \geq 0$, and $x \leq 1$.
- So, in the LP-relaxation, x is restricted to the interval $[0 \ldots 1]$.

  $\longrightarrow$ Due to the "surviving" constraints $x \geq 0$ and $x \leq 1$.
- Example TSP:
  - ◇ From Section 1.2, recall an ILP-formulation of the TSP.
  - ◇ The selection or rejection of a feature (i, j) amounts to an additional constraint: $X[i, j] = 1$ or $X[i, j] = 0$, respectively (using the notation "$X[\cdot, \cdot]$" from Section 1.2).

Example II: TSP

- There is a large body of literature on branch-and-bound on problem-specific relaxations and the induced lower bounds.
- In particular, many relaxation techniques have been developed that are tailored to the TSP.
- On the last slide, we have seen a first, simple example.
- On the next slide, we will briefly sketch another example.
- After that, we will consider one example in greater detail: 1-trees.

  $\longrightarrow$ A thorough discussion of the two examples on the last and next slides, respectively, would require too much additional mathematical background.

Next example for the TSP

- From Section 1.2, recall that the side constraints to avoid subtours are of exponential size (Slide 37).
- Forgetting about the problem of subtours yields a model of polynomial size ($\mathcal{O}(N^2)$).

  $\longrightarrow$ Just the objective function on Slide 35 and the constraints on Slide 36.
- This basic model may be solved quite efficiently.

  $\longrightarrow$ Not proved here (keyword b-*matching* in the literature).

Example: 1-tree relaxation

- This is one of the most prominent examples of relaxations for the TSP.
- *1-forest*: An undirected graph such that
  - ◇ the nodes are the N objects of the TSP instance and
  - ◇ the edges contain at most one cycle and this cycle contains at least one node with degree two.
- *1-tree*: An inclusion-maximal 1-forest.

  $\longrightarrow$ A 1-tree consists of a tree spanning V plus one additional edge, which closes a (unique) cycle.
- Consider the generic decision-tree definition for feature-based problems as before.
- Let v be some node of this tree, let X_v be the features already selected at v and Y_v the features already rejected at v.

Example II (TSP): 1–trees (cont'd)

- Consider the undirected complete graph $G = (V, E)$ on all N objects.
- Evidently, $(a, b) \in X_v$ implies $(b, a) \notin X_v$.

  $\longrightarrow$ No two directed arcs (a, b) of the original TSP correspond to the same (undirected) edge $\{a, b\}$ of G.
- Let $\tilde{X}_v$ and $\tilde{X}_w$ denote the sets of undirected edges of G corresponding this way to X_v and Y_v, respectively.
- Let $D[x, y]$ denote the distance from object x to object y.
- Transfer of the directed distances to the undirected graph G:

  For any two objects x and y, let

$$\tilde{D}(\{x, y\}) := \min\{D(x, y), D(y, x)\}.$$

Simple observations:

- The following set S_v of subsets E' of E is an independence system:

  It is $E' \in S_v$ if, and only if, two facts are true:
  $E' \subseteq E \setminus (\tilde{X}_v \cup \tilde{Y}_v)$ and $E' \cup \tilde{X}_v$ is a 1-forest.

- The value

$$\min \left\{ \sum_{e \in E' \cup \tilde{X}_v} \tilde{D}(e) \;\middle|\; E' \in S \right\}$$

  is a lower bound on the minimal length of a round tour in the original TSP instance.

Claim:

- From Section 3.2, example MST, recall that the set of all forests of an undirected graph $G = (V, E)$ is a matroid.
- It is easy to see (and proved analogously to Section 3.2) that the following set is a matroid, too:

  The set of all forests that cover X_v and avoid Y_v.

Consequence:

- The greedy algorithm computes a minimum spanning tree among all spanning trees that cover X_v and avoid Y_v.
- In particular, a minimal 1–tree that contains X_v and avoids Y_v can be computed efficiently.

$\longrightarrow$ Details on the next slide.

Algorithm for minimal 1-trees

- We choose a set $Z \subseteq V$ of nodes, $Z \neq \emptyset$
- For each node $z \in Z$, we compute the minimal 1-tree T_z in G such that z is on the (unique) cycle of T_z and has degree two in that 1-tree.
- Obviously, the length of each of these 1-trees with respect to $\tilde{D}(\cdot)$ is a lower bound on the minimal length of a round tour with respect to $D[\cdot, \cdot]$.

  $\longrightarrow$ Take the maximal length of all T_z, $z \in Z$, as the lower bound.
- How to compute T_z...

  $\longrightarrow$ On the next slide.

How to compute $\mathbf{T_z}$

- Apply the greedy algorithm for MST, but
  - ◇ excluding z and all edges that are incident to z and do not belong to $\tilde{X}_v$,
  - ◇ excluding all edges in $\tilde{Y}_v$, and
  - ◇ starting with the initial set $\tilde{X}_v$ instead of $\emptyset$.

  $\longrightarrow$ The result is a tree T'_z spanning $V \setminus \{z\}$ in G, plus zero, one, or two edges of $\tilde{X}_v$ that are incident to z.
- If less than two edges of $\tilde{X}_v$ are incident to z, add as many edges as necessary to T'_z such that z is incident to exactly two edges in T'_z, more specifically, choose the one or two cheapest ones.
- The result is T_z: T'_z plus $\tilde{X}_v$ plus two edges incident to z.

How to choose the set $\mathbf{Z}$

- The larger the set Z is, the tighter the lower bound will be.

  $\longrightarrow$ The smaller the number of decision-tree nodes to be explored will be.

- On the other hand, the smaller the set Z is, the smaller the run time will be to process a single decision-tree node.

- To optimize the total run time, a good balance must be found.

- No mathematical guidance is available for that.

  $\longrightarrow$ A good balance can only be determined empirically.

Proof of the claim:

- Let E_1 and E_2 be two members of this independence system such that $\#E_1 > \#E_2$
- We have to show: There is $e \in E_1 \setminus E_2$ such that $E_2 \cup \{e\}$ is a member of the independence system
- If E_2 is cycle–free, any edge $e \in E_1 \setminus E_2$ will do
- So assume that E_2 contains a (unique) cycle $\longrightarrow$ E_2 meets exactly as many as $\#E_2$ nodes
- On the other hand, E_1 meets at least as many as $\#E_1$ nodes
- Due to $\#E_1 > \#E_2$, there is at least one node met by E_1 but not by E_2
- An edge of E_1 meeting this node will do

Remarks on Example II:

- The chosen approach for constructing lower bounds may also affect the definition of the decision tree
- To see the problem, recall from Slide no. **??** that the nodes of the decision tree potentially stand for instances of an *extended* algorithmic problem
- What we need is an appropriate relaxation of this extended problem because the lower bound shall be applied at the tree nodes
- An appropriate relaxation of the original problem need not be appropriate for the extended problem
- Example II is a positive example of that:
  - ◇ The 1–tree problem was defined here in view of preceding decisions that enforced certain arcs
  - ◇ The resulting independence system is still a matroid

Section 3.6.3: Dynamic Programming

What is dynamic programming:

- In the branch-and-bound approach, the subtree rooted at a node v was excluded due to a comparison with the cost value of some feasible solution (or some upper bound).
- In the dynamic–programming approach, a subtree is excluded due to a comparison with another subtree.
- In many specific dynamic-programming algorithms in the literature, the tree nodes to be compared with each other are on the same height level of the decision tree. $\longrightarrow$ Our first example will demonstrate why.

Equivalence and domination

- Let v and w be two nodes of the decision tree.
- Let T_v and T_w denote the subtrees rooted at v and w, respectively.
- We say that v is
  - ◇ *equivalent* to w if the optimal values of feasible solutions in T_v and T_w are identical;
  - ◇ *dominated* by w if the optimal value in T_w is better than the optimal value in T_v.

Dynamic programming vs. branch-and-bound

- In the branch–and–bound approach, the subtree rooted at a node v was excluded because a lower bound ℓ_v on the optimal value in this subtree was not smaller than a certain upper bound u
- Typically, the upper bound was the objective value of some feasible solution s: $u = c[s]$
- Therefore, a subtree was discarded whenever it turned out that it was
  - ◇ *dominated* by s (if $c[s] < \ell_v$) or
  - ◇ *equivalent* to s (if $c[s] = \ell_v$)
- The idea of dynamic programming is to exclude subtrees
  - ◇ not if they are equivalent to/dominated by *leaves*
  - ◇ but if they are equivalent to/dominated by *other tree nodes*

Example I: One–dimensional knapsack packing

- Roughly speaking, the problem is to select items to be put in a knapsack such that all selected items fit into this knapsack. of the selected items is maximized.
- For simplicity, we will consider the one–dimensional case only:
  - ◇ *Input*: positive real numbers $a[1], \ldots, a[n]$, b.
  - ◇ *Feasible outputs*: selections $I \subseteq \{1, \ldots, n\}$ such that
$$\sum_{i \in I} a[i] \leq b\,.$$
  - ◇ *Objective*: maximizing $\sum_{i \in I} a[i]$.
- For example, this is the correct model if only a single dimension such as the volume or the weight is to be considered, not the exact shapes of the items nor of the knapsack.

Feature-based interpretation

- One-dimensional knapsack is naturally viewed as a feature–based problem:

  I is the selection of features from the ground set $\{1, \ldots, n\}$.

- *Decision tree* for the feature-based case: For $k \in \{0, \ldots, n\}$,
  - ◇ each node v on the k–th level of the decision tree represents a certain selection $I_v \subseteq \{1, \ldots, k\}$,
  - ◇ and for the two children of v (let's say w_1 and w_2) in the decision tree, it is
    - ▷ $I_{w_1} = I_v \cup \{k+1\}$ ("select $k+1$") and
    - ▷ $I_{w_2} = I_v$ ("reject $k+1$").

Claim:

Let v and w be two tree nodes one the same level. Suppose v and w are equivalent, that is,

$$\sum_{i \in I_v} a[i] = \sum_{i \in I_w} a[i] \,.$$

Then the subtree rooted at v *or* the subtree rooted at w may be cut off (but not both simultaneously, of course).

$\longrightarrow$ Proof on the next slide.

Proof of the claim:

- Each feasible solution in the subtree rooted at v (resp. w) may be written as $I_v \cup I'$ (resp. $I_w \cup I'$) for some

$$I' \subseteq \{k+1, \ldots, n\}\,.$$

- On the other hand, for each $I' \subseteq \{k+1, \ldots, n\}$, it is

$$\sum_{i \in I_v \cup I'} a[i] \;=\; \sum_{i \in I_w \cup I'} a[i]\,.$$

- *Consequence* for all $I' \subseteq \{k+1, \ldots, n\}$:
  - $\diamond$ $I_v \cup I'$ is feasible if, and only if, $I_w \cup I'$ is feasible.
  - $\diamond$ $I_v \cup I'$ is optimal if, and only if, $I_w \cup I'$ is optimal.

  $\longrightarrow$ There is an optimal solution in the subtree rooted at v if, and only if, there is an optimal solution in the subtree rooted at w.

Case of integral values a[1],. . . ,a[n]

- For $k \in \{0, \ldots, n\}$ and a node v of the decision tree on the k–th level, it is
$$I_v \in \left\{ 0, \ldots, \sum_{i=1}^{k} a[i] \right\}.$$

- For each node on the k–th level, two nodes on the $(k+1)$-st level are generated.

  $\longrightarrow$ One for "select" feature $k+1$,
  one for "reject" feature $k+1$.

- In summary, on the k-th level, at most $2 \cdot \sum_{i=1}^{k-1} a[i] + 2$ tree nodes are generated at all

  $\longrightarrow$ Of which at most $\sum_{i=1}^{k} a[i] + 1$ nodes will *not* be cut off.

Consequences for the integral case

- If $\sum_{i=1}^{n} a[i]$ is small, an optimal solution can be computed quite efficiently.

- In order to guarantee a small number of tree nodes on every level, the items should be sorted such that

$$a[1] \leq a[2] \leq \cdots \leq a[n] \,.$$

$\longrightarrow$ For every $k > 0$, this minimizes the value of

$$\sum_{i=1}^{k-1} a[i]$$

and thus the upper bound on the number of nodes generated on each level k.

Integrality and greatest common denominator

For d the greatest common denominator of the values $a[1], \ldots, a[n]$, we may replace b by $b/d[k]$ and $a[i]$ by $a[i]/d$ for each $i \in \{1, \ldots, n\}$.

$\longrightarrow$ For an efficient computation,

- it suffices that $\sum_{i=1}^{n} a[i]/d$ is small;
- a large value of $\sum_{i=1}^{n} a[i]$ is not a problem, then.

Dynamic programming and breadth-first search

- To exclude subtrees, we need all nodes on one level of the decision tree.

  $\longrightarrow$ This allows us to compare the decision-tree nodes on one level to determine cases of equivalence.

- Thus, breadth-first search seems to be the traversal strategy of choice.

- However, if the levels of the decision tree are intractably large, dynamic programming is infeasible.

  $\longrightarrow$ At least in this purist form – more sophisticated, specific variants may still be feasible in particular applications.

Example II: Shortest paths

- *Input*:
  - $G = (V, A)$ a strongly connected directed graph (the "network");
  - $\ell : A \longrightarrow \mathbb{R}_0^+$ (a "length" for each arc);
  - $s, t \in V$ ("source" and "target").
- *Feasible outputs*: paths from s to t.
- *Objective*: minimizing the length of the path, that is, the sum of the lengths of the arcs on this path.

Decision tree

- Each node u of the decision tree stands for a path p_u in G from s to some node $x[u] \in V$ (p_u may contain cycles).
- In turn, each such path is repesented by exactly one node of the decision tree.

  $\longrightarrow$ More formally, for each (s, v)-path in the network, $v \in V$, there is exactly one node u in the decision tree such that $x[u] = v$.

- For (v, w) be an arc of the network and u a decision-tree node such that $x[u] = v$, there is a corresponding arc (u, u') in the decision tree, and it is $x[u'] = w$.

  $\longrightarrow$ The arc (u, u') represents the option to extend p_u by (v, w), and the node u' represents the concatenation of p_u and (v, w).

Useless subtrees

- Let u_1 and u_2 be nodes of the decision tree such that $x[u_1] = x[u_2]$.
- Let p_{u_1} and p_{u_2} denote the paths corresponding to u_1 and u_2, respectively.
- Assume that
  - ◇ either p_{u_1} is shorter than p_{u_2}
  - ◇ or both p_{u_1} and p_{u_2} have the same length.
- Then we may safely cut off the subtree of the decision tree rooted at u_2.

Shortest paths and best–first search

- From Section 3.3, recall best–first search and the "goodness function".
- The other way round, we can define a "badness function" to guide a best–first search.

  $\longrightarrow$ In each stage, choose an arc with a minimal value of the badness function.
- For an arc (v, w) of the decision tree, we will take the length of the path p_w represented by w as the badness function value of (v, w).

Crucial fact

Since all ℓ–values are nonnegative, the arcs of the decision tree (that are not in discarded subtrees) are visited in the order of ascending badness function values.

$\longrightarrow$ Proof on the next slide.

Proof of the crucial fact

- Let (u, u') and (u', u'') be two arcs of the decision tree.

  $\longrightarrow$ Since all arc lengths in the network are nonnegative, the badness value of (u, u') cannot be larger than the badness value of (u', u'').

- Suppose for a contradiction that decision-tree arc (u_1, u_1') has a strictly smaller badness value than (u_2, u_2') but is visited later than (u_2, u_2').
- Due to the first point, all arcs on the path from the root to u_1 have strictly smaller badness values than (u_2, u_2').
- However, then all arcs on the path from the root to u_1' had been chosen prior to (u_2, u_2').

  $\longrightarrow$ Contradiction!

Cut-off strategy

Whenever we reach a node u_1 of the decision tree such that $x[u_1] = x[u_2]$ for a previously seen node u_2, we cut off the tree node rooted at u_1.

$\longrightarrow$ Very efficient: each arc (v, w) of the network is touched at most once because only the first visit of v will be continued by visiting the arcs that leave v.

Central claim

This cut-off strategy is conservative.

$\longrightarrow$ Proof on the next slide.

Proof of the central claim

- Let u_1 and u_2 be two nodes of the decision tree such that $x[u_1] = x[u_2]$ and u_1 has been visited before u_2.
- Let u_1' (resp. u_2') be the immediate predecessor of u_1 (resp. u_2) in the decision tree ($u_1' = u_2'$ is possible).
- The crucial fact from Slide 352 implies that the badness value of (u_1', u_1) is no worse than that of (u_2', u_2).
- Consequently, when (u_2', u_2) is visited by the search, u_2 may be cut off because the better (or equally good) option u_1 was seen before.

  $\longrightarrow$ Any extension of p_{u_1} towards t is at least as good as the same extension attached to p_{u_2}.

Remarks on Example II

- The algorithm presented here may be interpreted as a slight variation of *Dijkstra's algorithm*.

  $\longrightarrow$ In the standard form of Dijkstra's algorithm, the order of arrivals at a node is not necessarily ascending in the distances from the root.

  $\longrightarrow$ We have to update the distance at every arrival.

- Sometimes, our variation is presented as Dijkstra's algorithm in the literature.

Example III: Pareto shortest–paths

- We extend the shortest–path problem as follows:
  - ◇ there is no longer simply *one* arc length $\ell[\cdot]$,
  - ◇ but an arbitrary number $\ell_1[\cdot], \ldots, \ell_k[\cdot]$ of arc lengths,
  - ◇ and all of them shall be considered simultaneously for minimization.
- *Methodological problem*: in general, we cannot find an (s,t)–path that is minimal with respect to *all* arc lengths $\ell_i[\cdot]$.
- *Common solution*: a weaker notion of optimality – based on what is commonly called *equivalence*, *dominance* and *Pareto optimality*.

Equivalence and dominance

- *Equivalence*: Two (s,t)-paths p_1 and p_2 are said to be *equivalent* if $\ell_i[p_1] = \ell_i[p_2]$ for every $i \in \{1,\ldots,k\}$.
- *Dominance*: One (s,t)–path p_1 is said to *dominate* another (s,t)–path p_2 if
  - $\ell_i[p_1] \leq \ell_i[p_2]$ for *all* $i \in \{1,\ldots,k\}$ and
  - $\ell_i[p_1] < \ell_i[p_2]$ for *at least one* $i \in \{1,\ldots,k\}$.

Pareto optimality

- *Pareto optimality*: An (s,t)–path p is *Pareto optimal* if it is not dominated by any other (s,t)–path.
- Of course, only Pareto optimal solutions are of interest.

  $\longrightarrow$ Equivalence and dominance as defined on this slide is indeed an example of the general notions of equivalence and dominance as defined on Slide 339.

Crucial fact

The crucial fact from Slide no. 351 can be extended to this case:

- Let u_1 and u_2 be nodes of the decision tree such that $x[u_1] = x[u_2]$.
- Suppose that either p_{u_1} dominates p_{u_2} or p_{u_1} and p_{u_2} are equivalent.
- Then we may safely discard the subtree of the decision tree that is rooted at u_2.

Consequence:

At any node $v \in G$, we "only" have to maintain the Pareto–optimal (s, v)–paths.