Parallelism for Perturbation Management and Robust Plans: Playing the Repair Game

Sven Grothklags, Ulf Lorenz, and Jan Ehrhoff

2004
Parallelism for Perturbation Management
and Robust Plans:
Playing the Repair Game*

Sven Grothklags, Ulf Lorenz
University of Paderborn
Department of Computer Science
Fürstenallee 11, 33102 Paderborn, Germany

Jan Ehrhoff
Lufthansa Systems Airline Services GmbH
Network Management Solutions
Am Prime Parc 9, 65479 Raunheim, Germany

Abstract

An important insufficiency of modern industrial plans is their lack of robustness. Disruptions prevent companies from operating as planned before and induce high costs for trouble shooting. The main reason for the severe impact of disruptions stems from the fact that planners do traditionally consider deterministic input data to be available at planning time. In practice, however, there are often only distributions over the possible input data available.

The Repair Game is a formalization of a planning task, which brings two branches of computer science — game tree search and logistic planning optimization with OR tools — together. Playing it performs disruption management and generates robust plans with the help of game tree search. We present the game definition and a detailed motivation.

Technically, at each node of a search tree, a traditional optimization problem is solved such that large parts of the computation time are blocked by sequential computations. Nevertheless, there is enough node parallelism which we can make use of, in order to bring the running times onto a real-time level, and in order to increase the solution quality per minute significantly. This paper shows a convincing planning application which takes heavily advantage of parallel game tree search.

Experimental results are encouraging and indicate that our new method can significantly reduce disruption costs. Our method outperformed the traditional one by means of simulations, and the parallelization scales well.

*This work has been partially supported by the European Union within the 6th Framework Programme under contract 001907 (DELIS)
1 Introduction

Our new approach, the Repair Game, represents a generic methodology for general logistic planning tasks to incorporate stochastic input data. For the sake of clarity, we restrict the description mainly to the area of airline scheduling.

1.1 State of the Art

Planning in Airline Industry

Logistic planning teams use the whole spectrum of Operations Research methods in order to solve planning tasks using very large LP and IP models. These methods show an unequivocal success story in the area of all kinds of logistic planning.

Huge planning tasks are typically divided into sub-plans and super-plans such that a hierarchy arises. E.g., an airline planning process starts with the so called network design, which roughly tells the planning team which routes (so called legs) should be taken into account. Then, a first ‘plan’ is made which shows when which legs are offered to the customers. Thereafter, the planning process contains some layers which are of special interest for us.

Typically, airline companies have aircrafts of different types (so called subfleets), which differ in size and economic behavior. Given a flight schedule and a set of aircrafts, the fleet assignment problem is to determine which type of aircraft should fly each flight segment. A solution of the fleet assignment problem and the flight schedule together answers the question of how many aircrafts of which subfleets have to be at certain places at certain times. The fleet assignment problem is known to be NP-complete ([10]).

So called time-space networks, which are special flow graphs, can be used to give a specific mathematical programming formulation for this class of problems. They were introduced by Hane et al. in [11] to solve the fleet assignment problem and have been modified and extended by several authors [9, 13]. To solve the resulting min-cost flow problem on the time-space network it is typically transformed into a mixed integer linear program (MIP).

On the basis of the fleet assignment, a so called rotation plan is generated as a further step. The rotation plan describes exactly which physical aircraft must be at which place in the world and at which time. The rotation plan, generated out of the fleet assignment solution, is the final plan for the aircrafts which operation controllers try to follow. Further steps, which we do not deal with here, are crew rostering and crew pairing, which add crews to the aircrafts, and the scheduling of the ground handling.

Fleet assignment and rotation planning belong to long- and midterm planning phases. They produce airline plans according to economical parameters (passenger demands, revenues, costs, ...), airline parameters (existing aircraft types, capacities, crews, ...) and operational restrictions (maintenance times, flight durations, ...). The goal is to maximize the profitability. To enhance this profitability aircraft usage is more and more increased and airline plans become tighter and tighter. The tighter the plans become, the more frequently disruptions occur, which must be faced by the operation control management [20, 21]. Disruptions are events that prevent companies from operating as scheduled. They cause delays, aircraft changes, cancellations, ad hoc crew reassignment, slot problems, etc. In short, they cause lots of trouble and costs. Therefore, it is a
major desire of the long- and midterm planning groups to build plans which allow the operation control to go back to the original plan fast and without causing high costs.

**Game Tree Search**

Game tree search is the core of most attempts to make computers play games. The game tree acts as an error filter and examining the tree behaves similar to an approximation procedure. Computer chess, the most famous example for the effectiveness of game tree search, delivers an outstanding success story.

For most of the interesting board games, we do not know the correct evaluations of all positions. Therefore, we are forced to base our decisions on heuristic or vague knowledge.

![Figure 1. Only the (pre-)chosen partial tree is examined by a search algorithm.](image)

Typically, a game playing program consists of three parts: a move generator, which computes all possible moves in a given position; an evaluation procedure which implements a human expert’s knowledge about the value of a given position (these values are quite heuristic, fuzzy and limited) and a search algorithm, which organizes a forecast.

At some level of branching, the complete game tree (as defined by the rules of the game) is cut, the artificial leaves of the resulting subtree are evaluated with the heuristic evaluations, and these values are propagated to the root of the game tree as if they were real ones. For 2-person zero-sum games, computing this heuristic minimax value is by far the most successful approach in computer games history. Usually the tree to be searched is examined with the help of the Alphabeta algorithm [14], resp. with the Negascout variant [18]. Parallel machines are commonly used in order to speed up the search process [5].

When Shannon [23] proposed a design for a chess program in 1949 — which is in its core still used by all modern game playing programs — it seemed quite reasonable that deeper searches lead to better results. Theoretical analyzes [16, 17], however, showed the opposite. In 1988, I. Althöfer proposed a random tree model. In this model he was able to prove the usefulness of game tree search under certain conditions [1]. Anyway, the important observation over the last 40 years in the chess game and some other games is: the game tree acts as an error filter. Therefore, the faster and the more sophisticated the search algorithm, the better the search results! In 2002, this behavior could finally be explained with the help of a theoretical analysis about error propagation in game trees [15].

The approximation of the real root value with the help of fixed-depth searches, leads to good results. Nevertheless, several enhancements that form the search tree more individually have been found. Some of these techniques are domain independent like Singular Extensions [2], Nullmoves [4]. It is quite obvious that the form of the search tree strongly determines the quality of the search result.
Multistage Decisions under Risk

The reason for disruptions obviously stems from the fact that planners lack information about the real behavior of the environment at planning time. Often, data is not as fixed as assumed in the traditional planning process. Instead we know the data approximately, we know distributions over the data. In our airline example, we know e.g. a distribution over a leg’s possible arrival times. Traditionally, plans are built which maximize profits over ‘expected’ or just estimated input data. We believe that it is more realistic to optimize the expected payoff over all possible scenarios instead. This view on the world leads us to something that is often called ‘multistage decisions under risk’. It is a subtopic inside the large field of decision theoretic approaches (see e.g. [12]). Also linear stochastic programming [6, 19] shares this topic.

The appeal of our new approach, playing the so called Repair Game, lies in a slim problem description and in presenting solving techniques which provide the opportunity to plug in heuristics which quickly lead to good approximations, such that we are able to produce gains for large real world applications.

1.2 New Approach

Our approach can be can be described best by looking at a (stochastic) planning task in a ‘tree-wise’ manner. Let a tree $T$ be given that represents the possible scenarios as well as our possible actions in the forecast time-funnel. It consists of two different kinds of nodes, MIN nodes and AVG nodes. A node can be seen as a ‘system state’ at a certain point of time at which several alternative actions can be performed/scenarios can happen. Outgoing edges from MIN nodes represent our possible actions, outgoing edges from AVG nodes represent the ability of Nature to act in various ways. Every path from the root to a leaf can then be seen as a possible solution of our planning task; our actions are defined by the edges we take at MIN nodes under the condition that Natures acts as described by the edges that lead out of AVG nodes.

The Leaf values are supposed to be known and represent the total costs of the ‘planning path’ from the root to the leaf. The value of an inner MIN node is computed by taking the minimum of the values of its successors. The value of an inner AVG node is built by computing a weighted average of the values of its successor nodes. The weights correspond to realization probabilities of the scenarios.

Let a so called min-strategy $S$ be a subtree of $T$ which contains the root of $T$, and which contains exactly one successor at MIN nodes, and all successors that are in $T$ at AVG nodes. Each strategy $S$ shall have a value $f(S)$, defined as the value of $S$’s root. A principle variation $p(S)$, also called plan, of such a min-strategy can be determined by taking the edges of $S$ leaving the MIN nodes and a highest weighted outgoing edge of each AVG node. The connected path that contains the root is $p(S)$. We are interested in the plan $p(S_b)$ of the best strategy $S_b$ and in the expected costs $E(S_b)$ of $S_b$. The expected costs $E(p)$ of a plan $p$ are defined as the expected costs of the best strategy $S$ belonging to plan $p$, e.g. $E(p) = \min\{E(S) \mid p(S) = p\}$.

Because differences between planned operations and real operations cause costs, the expected costs associated with a given plan are not the same before and after the plan is distributed to customers. A plan gets a value of its own once it is published and other parties depend on it. At first glance this seems to complicate the given facts, but in truth this little detail enables our research to be practically relevant: We conjecture that the described problem to find a plan which
belongs to an optimal strategy is PSPACE-hard. Therefore we see no chance to find such an optimal plan for a full planning period of many days. As a consequence, we can only examine the possible future scenarios locally around a given point of time. This, however, implies that we need heuristic values for the artificial leaves of our search tree. A predefined, let us call it master plan, makes such a heuristic evaluation function available. We can measure how far a new plan is away from the master plan.

An analysis, as mentioned above, starts as soon as a disruption has occurred. We compute several partial plans which lead back to the original plan with low change-costs, examine further possible disruptions in the next time-step, compute further optional repair-plans, examine the next time-step etc., until we reach a search depth limited only by our computing power. The repaired plan that we select has small expected repair and future costs.

Our approach differs from known techniques in at least one of the following properties:

- We plan against the odds of an environment with the help of a local tree search into the future. When a disruption occurs, we compute several repair plans which lead back to the original plan, examine further possible disruptions in the next time-step, compute further repair-plans etc.
- We define robustness with the help of possible future events.
- We generate scenarios automatically and make profit from the bulk of scenarios. Generating this bulk of scenarios together with heuristics act as an approximation procedure.
- The possible actions of the environment need not to be equal in all branches of the forecast tree.

1.3 Organization of this paper

In this paper, we bring two branches of computer science — game tree search and logistic planning optimization with OR tools — together in order to produce better repair decisions, and more robust plans as well. The aim is to find repair alternatives (i.e. sub-plans) which minimize the expected damage in the future. We introduce the Repair Game as a reasonable formalization of the airline planning task on the level of disruption fighting. Section 2 describes the Repair Game, its formal definition, as well as an interpretation of the definition and an example. In Section 3 we describe a prototype, which produces robust repair decisions for disrupted airline schedules, on the basis of the Repair Game. Section 4 contains details of the parallelization of the search procedure. In Section 5 we compare the results of our new approach with an optimal repair procedure (in the traditional sense). A comparison of the sequential and the parallel version of our prototype is additionally given. Section 6 concludes.

2 The Repair Game

2.1 Definitions

We define the Repair Game via its game tree. Its examination gives us a measure for the robustness of a plan and it presents concrete operation recommendations as well.
Definition 2-1 (Game Tree)

For a rooted tree $T = (V, E)$ let $L(T) \subset V$ be the set of leaves of $T$. In this paper, a game tree $G = (V, E, h)$ is a rooted tree $(V, E)$, where the nodes are divided into three classes $V_{MAX} \cup V_{MIN} \cup V_{AVG} = V$ and every node has an associated value, defined by the function $h : V \rightarrow \mathbb{N}_0$. □

Nodes of a game tree $G$ represent positions of the underlying game, and edges move from one position to the next. The classes $V_{MAX}$, $V_{MIN}$, and $V_{AVG}$ represent three players $MAX$, $MIN$, and $AVG$ and for a node/position $v \in V_i$ the class $V_i$ determines the player $i$ who must perform the next move. The value of a game tree is defined by the so called *Minimax value of its root:

Definition 2-2 (*Minimax Value)

Let $G = (V, E, h)$ be a game tree and $w_v : N(v) \rightarrow [0, 1]$ be weight functions for all $v \in V_{AVG}$, where $N(v)$ is the set of all sons of a node $v$. The function $\ast\minmax : V \rightarrow \mathbb{N}_0$ is inductively defined by

$$\ast\minmax(v) := \begin{cases} h(v) & \text{if } v \in L(G) \\ \max\{\ast\minmax(v') | v' \in N(v)\} & \text{if } v \in V_{MAX} \setminus L(G) \\ \min\{\ast\minmax(v') | v' \in N(v)\} & \text{if } v \in V_{MIN} \setminus L(G) \\ \sum_{v' \in N(v)} (w_v(v') \cdot \ast\minmax(v')) & \text{if } v \in V_{AVG} \setminus L(G) \end{cases}$$

□

Definition 2-3 (Repair Game)

The goal of the Repair Game $=(G,p,g,f,s)$ is the calculation of $\ast\minmax(r)$ for a special game tree $G = (V, E, g + f)$ with root $r$ and uniform depth $t$; $p \in L(G)$ is a special leaf and $g, f$ and $s$ are functions. The game tree has the following properties:

- Let $P = (r = v_1, v_2, \ldots, p = v_t) \in V^t$ be the unique path from $r$ to $p$. $P$ describes a traditional, original plan.

- $V$ is partitioned into sets $S_1, \ldots, S_n, |V| \geq n \geq t$ by the function $s : V \rightarrow \{S_i\}_{1 \leq i \leq n}$. All nodes which belong to the same set $S_i$ are in the same state of the system — e.g. in aircraft scheduling: which aircraft is where at which point of time —, but they differ in the histories which have led them into this state.

- $g : \{S_i\}_{1 \leq i \leq n} \rightarrow \mathbb{N}_0$ defines the expected future costs for nodes depending on their state; for the special leaf $p$ holds $g(s(p)) = 0$

- $f : \bigcup_{1 \leq i \leq n} \{V\}^r \rightarrow \mathbb{N}_0$ defines the induced repair-costs for every possible (sub)path in $(V, E)$; every sub-path $P'$ of $P$ has zero repair-costs, $f(P') = 0$

- the node evaluation function $h : V \rightarrow \mathbb{N}_0$ is defined by $h(v) = g(s(v)) + f(r \ldots v)$; note that $h(p) = 0$ holds by the definition of $g$ and $f$
2.2 Interpretation and Airline Example

A planning team of, e.g. an airline company, may start the game with the construction of a traditional plan for its activities. The path $P$ represents this planned schedule, which also is the most expected plan in the time-funnel, and which interestingly gets an additional value of its own, as soon as it is generated. It is small, can be communicated, and as soon as a customer or a supplier has received the plan, each change of the plan means extra costs for the change. We take this plan as our master plan. Disruptions in airline transportation systems can now prevent airlines from executing their schedules as planned. Indeed, it is known that an aircraft needs approximately $x$ minutes from $A$ to $B$, but this 'approximation' includes a distribution over various possibilities. AVG-nodes are nature-nodes and the AVG-player formalizes this discretized distribution. A $MAX$-player will occur if we investigate the fact that airlines are in concurrency to other airlines, but we are not dealing with $MAX$-players in our current experiments. All the disruptions force the company to rearrange its schedule and to produce a new plan. The $MIN$-player represents the company itself. As soon as a specific disruption occurs, it will select a repairing sub-plan such that the repair costs plus the expected future repair costs are minimized.

As the value of a game tree leaf $v$ depends on how 'far' the path $(r, \ldots, v)$ is away from $P$, it is not possible to identify system states (where the aircrafts are at a specific time) with tree nodes. Therefore, the partition $V = S_1 \cup \ldots \cup S_n$. In $S_i$ all those nodes are collected which belong to the same system state, but have different histories, e.g. tree paths leading to them. All nodes with the same state $S_i$ have the same expected future costs. These costs are estimated by the function $g$. The function $f$ evaluates for an arbitrary partial path, how far it is away from the level-corresponding partial path of $P$. Inner nodes of the game tree are evaluated by the *Minimax function.

![Figure 2. The Repair Game Tree.](image)

Figure 2 shows a rotation plan at the right. Aircrafts A, B, and C are either on ground, or in the air, which is indicated by boxes. A shadowed box means that the original plan has been changed. The time goes from the top down. The left part of the figure shows a game tree, where
the leftmost path corresponds to the original plan \( P \). When a disruption occurs, we are forced to leave the plan, but hopefully we can return to it at some node \( v \). The fat path from node \( v \) downward consists of nodes with the same states as in the original plan. Thus, at node \( v \), we have costs for the path from the root \( r \) to \( v \), denoted by \( f(r \ldots v) \) and future expected costs, denoted by \( g(s(v)) \). If we follow the original plan from the beginning on, we will have the same expected costs at the time point represented by \( v \), but different path costs. The only node with zero costs typically is the special leaf node \( p \) of the original plan.

An objection against this kind of model may be that the tree \( G \) contains myriads of nodes. Nevertheless, we reply that this is not necessarily a reason to reject the model. It is estimated that there are about \( 10^{100} \) different nodes in chess, i.e. chess positions. Let us say, our game has (for realistic and interesting instances) \( 10^{10000} \) nodes. In both cases it will never be possible to examine the whole game tree. Nevertheless, we have seen in computer chess that it is possible to handle such a large state space successfully with the help of local selective search and heuristic node-evaluations. Although the future scenarios lead to potentially myriads of possibilities, we can organize a local search into the future after some disruption occurs.

3 The Repair Engines for Airline Experiments

In the following, we describe two repair engines which we compare with each other. The first one repairs a plan after a disruption with the help of a slightly modified time-space network such that the solution of the associated network flow problem is an optimal one, under the assumption that no more disruptions will ever occur. The second engine compares various solutions of the modified time-space network flow problem and examines various scenarios which might occur in the near future.

3.1 A Simple Repair Engine: The 'Myopic MIP' Solver

One possibility to find a repaired plan is to solve a mixed integer program which is similar to the one used for finding the original fleet assignment and rotation plan, but starting from the time point \( T \) with some changed constraints for the disrupted legs [11, 13]. This however, is not what planners aim at: The original rotation plan has a value of its own, because customers have anticipated the plan, as well as sub-contractors and others. The repair costs are mainly determined by the changes on the original plan, rather than by leg profits. Therefore, a new cost function is demanded, which minimizes the costs of the changes to get back to the original plan.

We model the problem of finding a repair solution as a min cost multi commodity integer network flow problem on an extended time-space network. We added the features of delays and cancellations to the model so that it is able to compute full-featured repair solutions.

In the MIP, corresponding to the modified time-space network, a set of binary decision variables \( x_{l,f} \) is associated with each leg \( l \), one variable for each subflet \( f \) that can operate leg \( l \); in a solution \( x_{l,f} \) is equal to one, if and only if leg \( l \) is operated by subflet \( f \). We omit further details here. They can be found in [11, 13]. The MIP based procedure which finds a new feasible plan, minimizes the costs for changes.

The basic time-space network does not support delays and cancellations of legs. We added these features to the model so that it is able to compute full-featured repair solutions. Canceling can
be incorporated by introducing an additional binary decision variable $x_{l, f}$ for every leg $l$ which is equal to one, if and only if leg $l$ is canceled. The ability to delay a leg is modeled in the following way. We restrict the freedom of delays to a (small) fixed number $D$ of possible delays, say 0, 30, 60 and 120 minutes ($D = 4$). Then, instead of using only one variable $x_{l, f}$ for a subfleet $f$ that can operate leg $l$, we introduce $D$ variables, one for each possible delay. The described extensions can be incorporated in a straightforward way into the time-space network. Moreover, using these variables in the objective function of the MIP allows us to directly minimize the costs of a repair alternative which mainly depends on the number of delays, reassignments and cancellations.

Thus, costs for changes are minimized, and the resulting plan is optimal under the assumption that no further disruptions will occur. Nevertheless, it is a myopic procedure within our time-dynamic model.

3.2 ‘T3’: An Engine which Plays the Repair Game

3.2.1 Forecasting Search Algorithm

A forecast procedure makes use of the dynamics locally around time $T$ and $T + d$ as follows: Instead of generating only one myopic MIP optimal solution for the recovery, we generate several ones. They are called our possible moves. A simple, certainly good heuristic is to demand that these solutions have nearly optimal costs concerning the cost function which minimizes the cost for changes. For all of these possible moves, we inspect what kind of relevant disruptions can happen within the next $d$ minutes. On all these scenarios, we repair the plan again with the help of the myopic MIP solver, which gives us value estimations for the arisen scenarios. At the moment, we experiment with only these depth 3 searches. We weight the scenarios according to their probabilities, and minimize over the expected values of the scenarios. Concerning the new part of the plan, we have not optimized the costs over expected input data, but we have approximately minimized the expected costs over possible scenarios.

Let $U$ and $L$ be the lower and the upper bound on all possible values of the game. The search algorithm consists of a modified Alphabeta algorithm [14], which has been extended by a nature-node branch. When nature has to move, each successor has a weight, and the values of all successors are summed up, concerning these weights. If we are at a nature-node $v$ and the first successors have lead to a value at node $v$, which can even in the best possible case not be brought below $\beta$, we have a cutoff. Of course, this is analogously valid for the $\alpha$ bound. The algorithm in figure 3 simplifies the one proposed in [3].

3.2.2 The Move Generator for the MIN-player

The move generator for the MIN-player computes several 'good' but diverse repair alternatives for a disrupted schedule. They all pass back into the original plan as cheaply and as quickly as possible. Each repair alternative can consist of a number of repair operations: delaying, reassigning and canceling of legs.

Basically, we use the myopic MIP of Section 3.1 to generate our repair alternatives. In order to get several good and diverse repair alternatives, we modified the Branch&Bound search that is used to solve the MIP. For CPLEX, this can be done by using callbacks, which give you full control on the tree search. Whenever a new integer solution (a good valid repair alternative),
value *minimax(node v, value α, value β)
1 generate all successors v₁, ..., vₖ of v
2 value val := 0;
3 if b = 0 return h(v) / * (leaf eval)*/
4 for i := 1 to b
5   if v is MAX-node {
6       α := max(α, *minimax(vᵢ, α, β))
7       if α ≥ β return α
8       if i = b return α
9   } else if v is MIN-node {
10      β := min(β, *minimax(vᵢ, α, β))
11     if α ≥ β return β
12     if i = b return β
13   } else {
14     let w₁, ..., wₖ be the weights of v₁, ..., vₖ
15     val += *minimax(vᵢ, α, β) · wᵢ;
16     if val + L · \( \sum_{j=i+1}^{b} w_j \geq β \) return β
17     if val + U · \( \sum_{j=i+1}^{b} w_j \leq α \) return α
18     if i = b return val
20 } 

Figure 3. Modified Alphabeta algorithm

that is close (e.g. 1%) to the current upper bound, is found, this solution is stored, but it is made
invalid for the MIP model by adding appropriate cuts. The Branch&Bound search terminates, as
soon as at least c repair alternatives are found, whereby at present we use c = 3.

In order to get diverse solutions out of this process, the added cuts must be chosen carefully.
We consider only the decision variables of legs, which are operated differently (delayed, reassigned
or canceled) than in the original schedule. We further group these variables by aircraft. For each
aircraft that operates k > 0 altered legs, we add one cut that sums up all decision variables of the
k altered legs (for the subfleet the aircraft belongs to), and we restrict this sum to be less than
or equal to k − 1. Note that for the current integer solution this sum is equal to k. Hence this
inequality cuts off the current integer solution from the solution space of the MIP.

3.2.3 The Move Generator for the AVG-player

At AVG-nodes, the average player examines which departures occur within the next time period
of 15 minutes. The atomic disruptions 'cancellation' and 'delay' of 30, 60, and 120 minutes are
generated for each leg, which departs during this time period. The atomic disruptions are weighted
according to their probabilities in the simulator. In principle, we would like to examine all these
atomic disruptions and all their combinations, but this leads to a number of scenarios that is much
too large. At the moment, we therefore restrict the search breadth by considering only atomic
disruptions.
4 Parallelization

4.1 Dynamic Load Balancing

The basic idea of our parallelization is to decompose the search tree, to search parts of the search tree in parallel and to balance the load dynamically with the help of the work stealing concept. This works as follows: First, a special processor \( P_0 \) gets the search problem and starts performing the negascout algorithm as if it would act sequentially. At the same time, the other processors send requests for work, the REQUEST message, to other randomly chosen processors. When a processor \( P_i \) that is already supplied with work catches such a request, it checks whether or not there are unexplored parts of its search tree, ready for evaluation. These unexplored parts are all rooted at the right siblings of the nodes of \( P_i \)'s search stack. Either \( P_i \) sends back that it cannot serve the work with the help of the NO-WORK message, or it sends such a node (a position in the search tree etc.) to the requesting processor \( P_j \). That is done with the help of the WORK message. Thus, \( P_i \) becomes a master itself, and \( P_j \) starts a sequential search on its own. The processors can be master and worker at the same time. The relationship is dynamically changed during the computation. When \( P_j \) has finished its work (possibly with the help of other processors), it sends an ANSWER message to \( P_i \). The master-worker relationship between \( P_i \) and \( P_j \) is released, and \( P_j \) is idle again. It again starts sending requests for work into the network.

When a processor \( P_i \) finds out that it has sent a wrong window to one of its workers \( P_j \), it makes a WINDOW message follow to \( P_j \). \( P_j \) stops its search, corrects the window and starts its old search from the beginning. If the message contained a cutoff, \( P_j \) just stops its work. A processor \( P_i \) can shorten its search stack from outside, when e.g. a CUTOFF message comes in which belongs to a node near the root. In absence of deep searches and advanced cutting techniques, however, CUTOFF and WINDOW messages are not yet used.

In many applications, the computation at each node is fast in relation to the exchange of a message. For example, there are chess programs which do not use more than 1000 processor cycles per search node, including move generation, moving and evaluating the node. In the application presented here, the situation is different. The sequential computations at each node takes several seconds if not even minutes, such that the latency of messages is not remarkably important. Instead, it is necessary to overlap communication and computations. We decoupled the MIP solver from the rest of the application and assigned a thread to it.

In distributed systems, messages can be delayed by the system. Messages from the past might arrive, which are outdated. Therefore, each message gets its own unique ID, such that we are always able to identify misleading results. We refer to [7] for further details.

4.2 Selection of Subproblems

The distributed subproblems are supposed to be large enough to keep the workers busy for a while. Otherwise, the communication overhead increases, since at least two messages (the subproblem itself and the result message) have to be exchanged for every subproblem. We only do not distribute work when the move which leads to the specific successor has the direct costs of zero, as then we know that no further work is necessary at that successor.
5 Experimental Results

Our experiments are performed on a cluster of the University of Paderborn. Each processor is a Pentium IV/2.8 GHz running the RedHat Linux operating system. The processors are connected by a Myrinet high speed interconnection network. Each node of the network consists of a dual computer.

5.1 Performance

In accordance with our industrial partner, we built a simulator in order to evaluate our model and the generic repair procedure, as described in the previous sections. Our simulator is less detailed than e.g. SimAir [22], but we believe that it is detailed enough to model the desired part of the reality. Furthermore, it is simple enough such that the occurring problems are computationally solvable.

First of all, we discretize the time of the rotation plan into steps of $d = 15$ minutes. Every departure in the current rotation plan is a possible event point, and all events inside one $d$ minute period are interpreted as simultaneous. When an event occurs, the leg which belongs to that event can be disrupted, i.e. it can be delayed by 30, 60, or 120 minutes, or it can be canceled with certain probabilities. Let $T$ be the present point of time. The simulator inspects the events between $T$ and $T + d$. The simulator informs a repair engine about the new disruptions, waits for a corrected rotation plan, and steps $d$ minutes forward.

The basis for our simulations a Lufthansa continental rotation plan plus the data which we need to build the plan and its repairs. The simulator has been designed with the help of our industrial partner. The plan consists of 20603 legs and was operated by 144 aircrafts belonging to 6 different subfleets. The MIP for this plan has 220460 columns, 99765 rows, and 580793 non-zeros. Partitioned into single days, the resulting partial plan for any single day consist of a corresponding smaller number of columns and non-zeros.

We simulated 14 days starting with Jan 3rd 1995, and we divided this time period into 14 pieces such that we arrived at a test set with 14 instances. Moreover, time is divided into segments of 15 minutes, and everything happening in one 15 minute block is assumed to be simultaneous. We compared the behavior of two repair engines: On the one hand we used a traditional repair engine, the 'myopic MIP', on the other hand we used our forecast algorithm, the Repair Game Engine 'T3', which was limited to a depth 3 search.

The interesting question was whether it is possible at all to generate alternatives to the myopic MIP outcome, which can be considered 'better' than the myopic outcome itself. Moreover it was not clear whether the differences would be measurable. Why should a (from a myopic point of view) suboptimal plan at time $T$ lead to better solutions when we can see only $d = 15$ minutes into the future? And even only with a simplified model in the search engine, compared with the disruption model in the simulator?

In the first run, the probabilities for a leg to be canceled/(delayed by 120 minutes)/(delayed by 60 minutes)/(delayed by 30 minutes) were set to 0.001/0.02/0.08/0.12. We counted the number of delay minutes (TIM), equipment changes\(^1\) (ECH) and cancellations (CNL) that the two repair engines generated during the simulation. Additionally we calculated how the revenue changed

\(^1\)The aircraft type of a leg has been changed.
Table 1. Simulation results for parameter set 0.001/0.02/0.08/0.12.

<table>
<thead>
<tr>
<th>MIP</th>
<th>TIM</th>
<th>ECH</th>
<th>revenue</th>
<th>c</th>
<th>T3</th>
<th>TIM</th>
<th>ECH</th>
<th>revenue</th>
<th>c</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>01/03</td>
<td>6270</td>
<td>0</td>
<td>0</td>
<td>313500</td>
<td>5850</td>
<td>0</td>
<td>0</td>
<td>292500</td>
<td>21000</td>
</tr>
<tr>
<td></td>
<td>01/04</td>
<td>6090</td>
<td>0</td>
<td>0</td>
<td>304500</td>
<td>6540</td>
<td>2</td>
<td>2219</td>
<td>344781</td>
<td>-40281</td>
</tr>
<tr>
<td></td>
<td>01/05</td>
<td>5820</td>
<td>0</td>
<td>0</td>
<td>291000</td>
<td>5910</td>
<td>0</td>
<td>0</td>
<td>295500</td>
<td>-4500</td>
</tr>
<tr>
<td></td>
<td>01/06</td>
<td>6240</td>
<td>2</td>
<td>-3717</td>
<td>335717</td>
<td>5910</td>
<td>2</td>
<td>-3717</td>
<td>319217</td>
<td>16500</td>
</tr>
<tr>
<td></td>
<td>01/07</td>
<td>5160</td>
<td>4</td>
<td>-28602</td>
<td>326602</td>
<td>5130</td>
<td>4</td>
<td>-28073</td>
<td>324573</td>
<td>2029</td>
</tr>
<tr>
<td></td>
<td>01/08</td>
<td>3600</td>
<td>4</td>
<td>-51674</td>
<td>271674</td>
<td>3570</td>
<td>4</td>
<td>-47784</td>
<td>266284</td>
<td>5390</td>
</tr>
<tr>
<td></td>
<td>01/09</td>
<td>5250</td>
<td>2</td>
<td>-28193</td>
<td>310693</td>
<td>5370</td>
<td>2</td>
<td>-6091</td>
<td>294591</td>
<td>16102</td>
</tr>
<tr>
<td></td>
<td>01/10</td>
<td>5730</td>
<td>0</td>
<td>0</td>
<td>285000</td>
<td>5940</td>
<td>0</td>
<td>0</td>
<td>297000</td>
<td>-10500</td>
</tr>
<tr>
<td></td>
<td>01/11</td>
<td>6270</td>
<td>4</td>
<td>-19915</td>
<td>373415</td>
<td>6090</td>
<td>4</td>
<td>-19915</td>
<td>364415</td>
<td>9000</td>
</tr>
<tr>
<td></td>
<td>01/12</td>
<td>6660</td>
<td>0</td>
<td>0</td>
<td>333000</td>
<td>6420</td>
<td>0</td>
<td>0</td>
<td>321000</td>
<td>12000</td>
</tr>
<tr>
<td></td>
<td>01/13</td>
<td>6750</td>
<td>4</td>
<td>-24790</td>
<td>402290</td>
<td>6810</td>
<td>2</td>
<td>-8782</td>
<td>369282</td>
<td>33008</td>
</tr>
<tr>
<td></td>
<td>01/14</td>
<td>5730</td>
<td>0</td>
<td>0</td>
<td>285000</td>
<td>5670</td>
<td>0</td>
<td>0</td>
<td>283500</td>
<td>3000</td>
</tr>
<tr>
<td></td>
<td>01/15</td>
<td>3390</td>
<td>0</td>
<td>0</td>
<td>169500</td>
<td>3270</td>
<td>0</td>
<td>0</td>
<td>163500</td>
<td>6000</td>
</tr>
<tr>
<td></td>
<td>01/16</td>
<td>5880</td>
<td>2</td>
<td>-4775</td>
<td>318775</td>
<td>5880</td>
<td>0</td>
<td>0</td>
<td>294000</td>
<td>24775</td>
</tr>
<tr>
<td>sums</td>
<td>78840</td>
<td>22</td>
<td>-161666</td>
<td>4323666</td>
<td>78360</td>
<td>20</td>
<td>-112143</td>
<td>4230143</td>
<td>93523</td>
<td></td>
</tr>
</tbody>
</table>

(mainly due to lost passengers). Motivated by our industrial partner, we used the cost function $c(TIM, ECH, CNL, revenue) = 50 \cdot TIM + 10000 \cdot ECH + 100000 \cdot CNL - revenue$ as objective function in the repair engines and to compare the outcome of the simulations.

The columns 2 to 5 of table 1 belong to the myopic MIP repair engine, and the columns 6 to 9 to the forecast algorithm. Cancellations (CNL) are not shown, because they were not produced in this specific run. The Δ-column gives us the gain according to our cost function c of T3 against the MIP solver. 11 of 14 days provided positive gain for the T3 engine. In the sum over all days, we gained 2.21 percent of the repair costs.

In order to compute a repair for one 15 minute period, the MIP solver on average used 8 seconds each. The T3 solver consumed about 200 times more time, as it on average has to evaluate 213 scenarios for each time step. Concerning its repair quality however, T3 could beat MIP in all three categories: TIM, ECH, and revenue. This is astonishing as we expected that T3 would sacrifice less valuable units (TIMs) in order to gain expected profit.

For a second simulation run, we changed the probabilities for a leg to be canceled/(delayed by 120 minutes)/(delayed by 60 minutes)/(delayed by 30 minutes) to 0.003/0.04/0.16/0.24. The results can be found in table 2.

Again, T3 outperformed the MIP solver on 10 of 14 days. Altogether, we gained 3.35 percent of the repair costs. Here we can observe that T3 produced less of the costly cancellations but instead generated more equipment changes. Quite an interesting observation is that we can detect Sundays on Jan 8th and Jan 15th. Disruptions have significantly less consequences on these days as the schedules are not as tight as during the rest of the week.
Table 2. Simulation results for parameter set 0.003/0.04/0.16/0.24.

<table>
<thead>
<tr>
<th># proc.</th>
<th>Jan 3rd simulation time</th>
<th>Jan 4th simulation time</th>
<th>Jan 5th simulation time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>speedup</td>
<td>speedup</td>
<td>speedup</td>
</tr>
<tr>
<td>1</td>
<td>226057</td>
<td>1</td>
<td>219039</td>
</tr>
<tr>
<td>2</td>
<td>128608</td>
<td>1.76</td>
<td>126915</td>
</tr>
<tr>
<td>4</td>
<td>68229</td>
<td>3.31</td>
<td>66281</td>
</tr>
<tr>
<td>8</td>
<td>46675</td>
<td>4.84</td>
<td>46065</td>
</tr>
</tbody>
</table>

Table 3. Speedup results

5.2 Speedups

We measure speedups of our program with the help of the first three days of the test set which consists of the 14 single days, mentioned above. The time for simulation of the days using one processor is compared with the running time of several processors. The speedup is the sum of the times of the sequential version divided by the sum of the times of a parallel version. Concerning [8], this is the correct way for measuring speedups. Each test day consists of 96 single measure points. As the sequential and the parallel version indeed do exactly the same, we need not examine search overhead. Neither the costs for communication are relevant. The only reason that we cannot achieve the full speedup are the large sequential computing periods, caused by the commercial MIP solver. This seems unavoidable, as long as there is no efficient parallel MIP solver.

In Table 3 the simulation times (in seconds) and the resulting speedups are shown for 3 different days of our data set. For each day we ran our simulation on 1, 2, 4 and 8 processors. The speedups for all three days are very similar to each other. Therefore, we think that the data basis for our speedup experiments is large enough. Our program scales quite well up to 4 processors. With 8
processors we can still improve the overall running time, but the efficiency is worse compared to fewer processors.

6 Conclusion and Future Work

Playing the Repair Game led to more robust (sub-)plans in airline scheduling than a myopic MIP solver could provide. Our forecast strategy outperformed the myopic MIP solver by means of simulations. First experiments in the area of airline planning are encouraging. We have parallelized the search in order to drop the computation times to real time. Next, we will look for more clever and selective search heuristics, examine heuristics which give us fast new moves, and refine the simulator.

We answered one of the most aggressive questions to parallel computing: 'what is it good for?' with an application which operates in the area of Operations Research and that makes profit from its speed. The faster the application can be performed, the larger is the gained profit.

References


