Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison

CHRISTIAN BLUM

Université Libre de Bruxelles

AND

ANDREA ROLI

Università degli Studi di Bologna

The field of metaheuristics for the application to combinatorial optimization problems is a rapidly growing field of research. This is due to the importance of combinatorial optimization problems for the scientific as well as the industrial world. We give a survey of the nowadays most important metaheuristics from a conceptual point of view. We outline the different components and concepts that are used in the different metaheuristics in order to analyze their similarities and differences. Two very important concepts in metaheuristics are intensification and diversification. These are the two forces that largely determine the behavior of a metaheuristic. They are in some way contrary but also complementary to each other. We introduce a framework, that we call the *I&D* frame, in order to put different intensification and diversification components into relation with each other. Outlining the advantages and disadvantages of different metaheuristic approaches we conclude by pointing out the importance of hybridization of metaheuristics as well as the integration of metaheuristics and other methods for optimization.

Categories and Subject Descriptors: G.2.1 [**Discrete Mathematics**]: Combinatorics—*combinatorial algorithms*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*heuristic methods*

General Terms: Algorithms

Additional Key Words and Phrases: Metaheuristics, combinatorial optimization, intensification, diversification.

C. Blum acknowledges support by the "Metaheuristics Network," a Research Training Network funded by the Improving Human Potential program of the CEC, contract HPRN-CT-1999-00106.

A. Roli acknowledges support by the CEC through a "Marie Curie Training Site" fellowship, contract HPMT-CT-2000-00032.

The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication.

Authors' addresses: C. Blum, Université Libre de Bruxelles, IRIDIA, Avenue Franklin Roosevelt 50, CP 194/6, 1050 Brussels, Belgium; email: cblum@ulb.ac.be; A. Roli, DEIA—Università degli Studi di Bologna, Viale Risorgimento, 2-Bologna, Italy; email: aroli@deis.unibo.it.

1. INTRODUCTION

Many optimization problems of practical as well as theoretical importance consist of the search for a "best" configuration of a set of variables to achieve some goals. They seem to divide naturally into two categories: those where solutions are encoded with *real-valued* variables, and those where solutions are encoded with *discrete* variables. Among the latter ones we find a class of problems called Combinatorial Optimization (CO) problems. According to Papadimitriou and Steiglitz [1982], in CO problems, we are looking for an object from a finite—or possibly countably infinite—set. This object is typically an integer number, a subset, a permutation, or a graph structure.

Definition 1.1. A *Combinatorial Optimization* problem $P = (S, f)$ can be defined by:

—a set of variables $X = \{x_1, \ldots, x_n\}$;
—variable domains $D_1, \ldots, D_n$;
—constraints among variables;
—an *objective function* f to be minimized,[1] where $f : D_1 \times \cdots \times D_n \rightarrow \mathbb{R}^+$;

The set of all possible feasible assignments is

$$S = \{s = \{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in D_i, s \text{ satisfies all the constraints}\}.$$

S is usually called a *search (or solution) space*, as each element of the set can be seen as a candidate solution. To solve a combinatorial optimization problem one has to find a solution $s^* \in S$ with minimum objective function value, that is, $f(s^*) \leq f(s)\ \forall s \in S$. s^* is called a globally optimal solution of (S, f) and the set $S^* \subseteq S$ is called the set of globally optimal solutions.

Examples for CO problems are the Travelling Salesman problem (TSP), the Quadratic Assignment problem (QAP), Timetabling and Scheduling problems. Due to the practical importance of CO problems, many algorithms to tackle them have been developed. These algorithms can be classified as either *complete* or *approximate* algorithms. Complete algorithms are guaranteed to find for every finite size instance of a CO problem an optimal solution in bounded time (see Papadimitriou and Steiglitz [1982] and Nemhauser and Wolsey [1988]). Yet, for CO problems that are $\mathcal{NP}$-hard [Garey and Johnson 1979], no polynomial time algorithm exists, assuming that $\mathcal{P} \neq \mathcal{NP}$. Therefore, complete methods might need exponential computation time in the worst-case. This often leads to computation times too high for practical purposes. Thus, the use of approximate methods to solve CO problems has received more and more attention in the last 30 years. In approximate methods we sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in a significantly reduced amount of time.

Among the basic approximate methods we usually distinguish between *constructive* methods and *local search* methods. Constructive algorithms generate solutions from scratch by adding—to an initially empty partial solution—components, until a solution is complete. They are typically the fastest approximate methods, yet they often return solutions of inferior quality when compared to local search algorithms. Local search algorithms start from some initial solution and iteratively try to replace the current solution by a better solution in an appropriately defined neighborhood of the current solution, where the neighborhood is formally defined as follows:

Definition 1.2. A *neighborhood structure* is a function $\mathcal{N} : S \rightarrow 2^S$ that assigns to every $s \in S$ a set of neighbors $\mathcal{N}(s) \subseteq S$. $\mathcal{N}(s)$ is called the neighborhood of s.

The introduction of a neighborhood structure enables us to define the concept of *locally* minimal solutions.

Definition 1.3. A *locally minimal solution (or local minimum)* with respect to a neighborhood structure $\mathcal{N}$ is a solution $\hat{s}$ such that $\forall\, s \in \mathcal{N}(\hat{s}) : f(\hat{s}) \leq f(s)$. We

[1] As maximizing an objective function f is the same as minimizing $-f$, in this work we will deal, without loss of generality, with minimization problems.

call $\hat{s}$ a strict locally minimal solution if $f(\hat{s}) < f(s)\ \forall s \in \mathcal{N}(\hat{s})$.

In the last 20 years, a new kind of approximate algorithm has emerged which basically tries to combine basic heuristic methods in higher level frameworks aimed at efficiently and effectively exploring a search space. These methods are nowadays commonly called *metaheuristics*.[2] The term *metaheuristic*, first introduced in Glover [1986], derives from the composition of two Greek words. *Heuristic* derives from the verb *heuriskein* ($\epsilon\upsilon\rho\iota\sigma\kappa\epsilon\iota\nu$) which means "to find", while the suffix *meta* means "beyond, in an upper level". Before this term was widely adopted, metaheuristics were often called *modern heuristics* [Reeves 1993].

This class of algorithms includes[3]—but is not restricted to—Ant Colony Optimization (ACO), Evolutionary Computation (EC) including Genetic Algorithms (GA), Iterated Local Search (ILS), Simulated Annealing (SA), and Tabu Search (TS). Up to now there is no commonly accepted definition for the term metaheuristic. It is just in the last few years that some researchers in the field tried to propose a definition. In the following we quote some of them:

"A metaheuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions." [Osman and Laporte 1996].

"A metaheuristic is an iterative master process that guides and modifies the operations of subordinate heuristics to efficiently produce high-quality solutions. It may manipulate a complete (or incomplete) single solution or a collection of solutions at each iteration. The subordinate heuristics may be high (or low) level procedures, or a simple local search, or just a construction method." [Voß et al. 1999].

"Metaheuristics are typically high-level strategies which guide an underlying, more problem specific heuristic, to increase their performance. The main goal is to avoid the disadvantages of iterative improvement and, in particular, multiple descent by allowing the local search to escape from local optima. This is achieved by either allowing worsening moves or generating new starting solutions for the local search in a more "intelligent" way than just providing random initial solutions. Many of the methods can be interpreted as introducing a bias such that high quality solutions are produced quickly. This bias can be of various forms and can be cast as descent bias (based on the objective function), memory bias (based on previously made decisions) or experience bias (based on prior performance). Many of the metaheuristic approaches rely on probabilistic decisions made during the search. But, the main difference to pure random search is that in metaheuristic algorithms randomness is not used blindly but in an intelligent, biased form." [Stützle 1999b].

"A metaheuristic is a set of concepts that can be used to define heuristic methods that can be applied to a wide set of different problems. In other words, a metaheuristic can be seen as a general algorithmic framework which can be applied to different optimization problems with relatively few modifications to make them adapted to a specific problem." [Metaheuristics Network Website 2000].

Summarizing, we outline fundamental properties which characterize metaheuristics:

- —Metaheuristics are strategies that "guide" the search process.
- —The goal is to efficiently explore the search space in order to find (near-) optimal solutions.
- —Techniques which constitute metaheuristic algorithms range from simple local search procedures to complex learning processes.

[2]The increasing importance of metaheuristics is underlined by the biannual Metaheuristics International Conference (MIC). The 5th is being held in Kyoto in August 2003 (*http://www-or.amp.i.kyoto-u.ac.jp/mic2003/*).

[3]In alphabetical order.

—Metaheuristic algorithms are approximate and usually non-deterministic.

—They may incorporate mechanisms to avoid getting trapped in confined areas of the search space.

—The basic concepts of metaheuristics permit an abstract level description.

—Metaheuristics are not problem-specific.

—Metaheuristics may make use of domain-specific knowledge in the form of heuristics that are controlled by the upper level strategy.

—Todays more advanced metaheuristics use search experience (embodied in some form of memory) to guide the search.

In short, we could say that metaheuristics are high level strategies for exploring search spaces by using different methods. Of great importance hereby is that a dynamic balance is given between *diversification* and *intensification*. The term diversification generally refers to the exploration of the search space, whereas the term intensification refers to the exploitation of the accumulated search experience. These terms stem from the Tabu Search field [Glover and Laguna 1997] and it is important to clarify that the terms *exploration* and *exploitation* are sometimes used instead, for example in the Evolutionary Computation field [Eiben and Schippers 1998], with a more restricted meaning. In fact, the notions of exploitation and exploration often refer to rather short-term strategies tied to randomness, whereas intensification and diversification also refer to medium- and long-term strategies based on the usage of memory. The use of the terms diversification and intensification in their initial meaning becomes more and more accepted by the whole field of metaheuristics. Therefore, we use them throughout the article. The balance between diversification and intensification as mentioned above is important, on one side to quickly identify regions in the search space with high quality solutions and on the other side not to waste too much time in regions of the search space which are either already explored or which do not provide high quality solutions.

The search strategies of different metaheuristics are highly dependent on the philosophy of the metaheuristic itself. Comparing the strategies used in different metaheuristics is one of the goals of Section 5. There are several different philosophies apparent in the existing metaheuristics. Some of them can be seen as "intelligent" extensions of local search algorithms. The goal of this kind of metaheuristic is to escape from local minima in order to proceed in the exploration of the search space and to move on to find other hopefully better local minima. This is for example the case in Tabu Search, Iterated Local Search, Variable Neighborhood Search, GRASP and Simulated Annealing. These metaheuristics (also called trajectory methods) work on one or several neighborhood structure(s) imposed on the members (the solutions) of the search space.

We can find a different philosophy in algorithms like Ant Colony Optimization and Evolutionary Computation. They incorporate a learning component in the sense that they implicitly or explicitly try to learn correlations between decision variables to identify high quality areas in the search space. This kind of metaheuristic performs, in a sense, a biased sampling of the search space. For instance, in Evolutionary Computation this is achieved by recombination of solutions and in Ant Colony Optimization by sampling the search space in every iteration according to a probability distribution.

The structure of this work is as follows: There are several approaches to classify metaheuristics according to their properties. In Section 2, we briefly list and summarize different classification approaches. Section 3 and Section 4 are devoted to a description of the most important metaheuristics nowadays. Section 3 describes the most relevant trajectory methods and, in Section 4, we outline population-based methods. Section 5 aims at giving a unifying view on metaheuristics with respect to the way they achieve intensification and diversification. This is done by the

introduction of a unifying framework, the *I&D frame*. Finally, Section 6 offers some conclusions and an outlook to the future.

We believe that it is hardly possible to produce a completely accurate survey of metaheuristics that is doing justice to every viewpoint. Moreover, a survey of an immense area such as metaheuristics has to focus on certain aspects and therefore has unfortunately to neglect other aspects. Therefore, we want to clarify at this point that this survey is done from the conceptual point of view. We want to outline the different concepts that are used in different metaheuristics in order to analyze the similarities and the differences between them. We do not go into the implementation of metaheuristics, which is certainly an important aspect of metaheuristics research with respect to the increasing importance of efficiency and software reusability. We refer the interested reader to Whitley [1989], Grefenstette [1990], Fink and Voß [1999], Schaerf et al. [2000], and Voß and Woodruff [2002].

2. CLASSIFICATION OF METAHEURISTICS

There are different ways to classify and describe metaheuristic algorithms. Depending on the characteristics selected to differentiate among them, several classifications are possible, each of them being the result of a specific viewpoint. We briefly summarize the most important ways of classifying metaheuristics.

Nature-inspired vs. non-nature inspired. Perhaps, the most intuitive way of classifying metaheuristics is based on the origins of the algorithm. There are nature-inspired algorithms, like Genetic Algorithms and Ant Algorithms, and non nature-inspired ones such as Tabu Search and Iterated Local Search. In our opinion this classification is not very meaningful for the following two reasons. First, many recent hybrid algorithms do not fit either class (or, in a sense, they fit both at the same time). Second, it is sometimes difficult to clearly attribute an algorithm to one of the two classes. So, for example, one might ask the question if the use of memory in Tabu Search is not nature-inspired as well.

Population-based vs. single point search. Another characteristic that can be used for the classification of metaheuristics is the number of solutions used at the same time: Does the algorithm work on a population or on a single solution at any time? Algorithms working on single solutions are called *trajectory methods* and encompass local search-based metaheuristics, like Tabu Search, Iterated Local Search and Variable Neighborhood Search. They all share the property of describing a trajectory in the search space during the search process. Population-based metaheuristics, on the contrary, perform search processes which describe the evolution of a set of points in the search space.

Dynamic vs. static objective function. Metaheuristics can also be classified according to the way they make use of the objective function. While some algorithms keep the objective function given in the problem representation "as it is", some others, like Guided Local Search (GLS), modify it during the search. The idea behind this approach is to escape from local minima by modifying the search landscape. Accordingly, during the search the objective function is altered by trying to incorporate information collected during the search process.

One vs. various neighborhood structures. Most metaheuristic algorithms work on one single neighborhood structure. In other words, the fitness landscape topology does not change in the course of the algorithm. Other metaheuristics, such as Variable Neighborhood Search (VNS), use a set of neighborhood structures which gives the possibility to diversify the search by swapping between different fitness landscapes.

Memory usage vs. memory-less methods. A very important feature to classify metaheuristics is the use they make of the search history, that is, whether they use memory or not.[4] Memory-less algorithms

[4]Here we refer to the use of adaptive memory, in contrast to rather rigid memory, as used for instance in Branch & Bound.

perform a Markov process, as the information they exclusively use to determine the next action is the current state of the search process. There are several different ways of making use of memory. Usually we differentiate between the use of short term and long term memory. The first usually keeps track of recently performed moves, visited solutions or, in general, decisions taken. The second is usually an accumulation of synthetic parameters about the search. The use of memory is nowadays recognized as one of the fundamental elements of a powerful metaheuristic.

In the following, we describe the most important metaheuristics according to the single point vs. population-based search classification, which divides metaheuristics into trajectory methods and population-based methods. This choice is motivated by the fact that this categorization permits a clearer description of the algorithms. Moreover, a current trend is the hybridization of methods in the direction of the integration of single point search algorithms in population-based ones. In the following two sections, we give a detailed description of nowadays most important metaheuristics.

3. TRAJECTORY METHODS

In this section we outline metaheuristics called *trajectory methods*. The term trajectory methods is used because the search process performed by these methods is characterized by a trajectory in the search space. Hereby, a successor solution may or may not belong to the neighborhood of the current solution.

The search process of trajectory methods can be seen as the evolution in (discrete) time of a discrete dynamical system [Bar-Yam 1997; Devaney 1989]. The algorithm starts from an initial state (the initial solution) and describes a trajectory in the state space. The system dynamics depends on the strategy used; simple algorithms generate a trajectory composed of two parts: a *transient* phase followed by an *attractor* (a fixed point, a cycle or a complex attractor). Algorithms with advanced strategies generate more complex

```
s ← GenerateInitialSolution()
repeat
    s ← Improve(N(s))
until no improvement is possible
```

Fig. 1. Algorithm: Iterative Improvement.

trajectories which can not be subdivided in those two phases. The characteristics of the trajectory provide information about the behavior of the algorithm and its effectiveness with respect to the instance that is tackled. It is worth underlining that the dynamics is the result of the combination of algorithm, problem representation and problem instance. In fact, the problem representation together with the neighborhood structures define the search landscape; the algorithm describes the strategy used to explore the landscape and, finally, the actual search space characteristics are defined by the problem instance to be solved.

We will first describe basic local search algorithms, before we proceed with the survey of more complex strategies. Finally, we deal with algorithms that are general explorative strategies which may incorporate other trajectory methods as components.

3.1. Basic Local Search: Iterative Improvement

The basic local search is usually called *iterative improvement*, since each move[5] is only performed if the resulting solution is better than the current solution. The algorithm stops as soon as it finds a local minimum. The high level algorithm is sketched in Figure 1.

The function Improve($\mathcal{N}(s)$) can be in the extremes either a *first improvement*, or a *best improvement* function, or any intermediate option. The former scans the neighborhood $\mathcal{N}(s)$ and chooses the first solution that is better than s, the latter exhaustively explores the neighborhood and returns one of the solutions with the lowest objective function value. Both methods

[5] A move is the choice of a solution s' from the neighborhood $\mathcal{N}(s)$ of a solution s.

```
s ← GenerateInitialSolution()
T ← T_0
while termination conditions not met do
  s' ← PickAtRandom(N(s))
  if (f(s') < f(s)) then
    s ← s'          % s' replaces s
  else
    Accept s' as new solution with probability p(T, s', s)
  endif
  Update(T)
endwhile
```

Fig. 2. Algorithm: Simulated Annealing (SA).

stop at local minima. Therefore, their performance strongly depends on the definition of $\mathcal{S}$, f and $\mathcal{N}$. The performance of iterative improvement procedures on CO problems is usually quite unsatisfactory. Therefore, several techniques have been developed to prevent algorithms from getting trapped in local minima, which is done by adding mechanisms that allow them to escape from local minima. This also implies that the termination conditions of metaheuristic algorithms are more complex than simply reaching a local minimum. Indeed, possible termination conditions include: maximum CPU time, a maximum number of iterations, a solution s with $f(s)$ less than a predefined threshold value is found, or the maximum number of iterations without improvements is reached.

3.2. Simulated Annealing

Simulated Annealing (SA) is commonly said to be the oldest among the metaheuristics and surely one of the first algorithms that had an explicit strategy to escape from local minima. The origins of the algorithm are in statistical mechanics (Metropolis algorithm) and it was first presented as a search algorithm for CO problems in Kirkpatrick et al. [1983] and Cerny [1985]. The fundamental idea is to allow moves resulting in solutions of worse quality than the current solution (uphill moves) in order to escape from local minima. The probability of doing such a move is decreased during the search. The high level algorithm is described in Figure 2.

The algorithm starts by generating an initial solution (either randomly or heuristically constructed) and by initializing the so-called temperature parameter T. Then, at each iteration a solution $s' \in \mathcal{N}(s)$ is randomly sampled and it is accepted as new current solution depending on $f(s)$, $f(s')$ and T. s' replaces s if $f(s') < f(s)$ or, in case $f(s') \geq f(s)$, with a probability which is a function of T and $f(s') - f(s)$. The probability is generally computed following the Boltzmann distribution $\exp(-\frac{f(s')-f(s)}{T})$.

The temperature T is decreased[6] during the search process, thus at the beginning of the search the probability of accepting uphill moves is high and it gradually decreases, converging to a simple iterative improvement algorithm. This process is analogous to the annealing process of metals and glass, which assume a low energy configuration when cooled with an appropriate cooling schedule. Regarding the search process, this means that the algorithm is the result of two combined strategies: random walk and iterative improvement. In the first phase of the search, the bias toward improvements is low and it permits the exploration of the search space; this erratic component is slowly decreased thus leading the search to converge to a (local) minimum. The probability of accepting uphill moves is controlled by two factors: the difference of the objective functions and the temperature. On the one hand, at fixed temperature, the higher the difference $f(s')-f(s)$, the lower the probability to accept a move from s to s'. On the other hand, the higher T, the higher the probability of uphill moves.

The choice of an appropriate cooling schedule is crucial for the performance of the algorithm. The cooling schedule defines the value of T at each iteration k, $T_{k+1} = Q(T_k, k)$, where $Q(T_k, k)$ is a function of the temperature and of the iteration number. Theoretical results on nonhomogeneous Markov chains [Aarts et al. 1997] state that under particular conditions on the cooling schedule, the algorithm converges in probability to a global

[6] T is not necessarily decreased in a monotonic fashion. Elaborate cooling schemes also incorporate an occasional increase of the temperature.

minimum for $k \to \infty$. More precisely:

$$\exists \Gamma \in \mathbb{R} \text{ such that } \lim_{k\to\infty} \mathbf{p}(\text{global minimum found after } k \text{ steps}) = 1$$
$$\text{iff} \quad \sum_{k=1}^{\infty} \exp\left(\frac{\Gamma}{T_k}\right) = \infty$$

A particular cooling schedule that fulfils the hypothesis for the convergence is the one that follows a logarithmic law: $T_{k+1} = \frac{\Gamma}{\log(k+k_0)}$ (where k_0 is a constant). Unfortunately, cooling schedules which guarantee the convergence to a global optimum are not feasible in applications, because they are too slow for practical purposes. Therefore, faster cooling schedules are adopted in applications. One of the most used follows a geometric law: $T_{k+1} = \alpha T_k$, where $\alpha \in (0, 1)$, which corresponds to an exponential decay of the temperature.

The cooling rule may vary during the search, with the aim of tuning the balance between diversification and intensification. For example, at the beginning of the search, T might be constant or linearly decreasing, in order to sample the search space; then, T might follow a rule such as the geometric one, to converge to a local minimum at the end of the search. More successful variants are *nonmonotonic* cooling schedules (e.g., see Osman [1993] and Lundy and Mees [1986]). Nonmonotonic cooling schedules are characterized by alternating phases of cooling and reheating, thus providing an oscillating balance between diversification and intensification.

The cooling schedule and the initial temperature should be adapted to the particular problem instance, since the cost of escaping from local minima depends on the structure of the search landscape. A simple way of empirically determining the starting temperature T_0 is to initially sample the search space with a random walk to roughly evaluate the average and the variance of objective function values. But also more elaborate schemes can be implemented [Ingber 1996].

The dynamic process described by SA is a *Markov chain* [Feller 1968], as it follows a trajectory in the state space in which the successor state is chosen depending only on the incumbent one. This means that basic SA is memory-less. However, the use of memory can be beneficial for SA approaches (see, e.g., Chardaire et al. [1995]).

```
s ← GenerateInitialSolution()
TabuList ← ∅
while termination conditions not met do
  s ← ChooseBestOf(N(s) \ TabuList)
  Update(TabuList)
endwhile
```

Fig. 3. Algorithm: Simple Tabu Search (TS).

SA has been applied to several CO problems, such as the Quadratic Assignment Problem (QAP) [Connolly 1990] and the Job Shop Scheduling (JSS) problem [Van Laarhoven et al. 1992]. References to other applications can be found in Aarts and Lenstra [1997], Ingber [1996] and Fleischer [1995]. SA is nowadays used as a component in metaheuristics, rather than applied as stand-alone search algorithm. Variants of SA called Threshold Accepting and The Great Deluge Algorithm were presented by Dueck and Scheuer [1990] and Dueck [1993].

3.3. Tabu Search

Tabu Search (TS) is among the most cited and used metaheuristics for CO problems. TS basic ideas were first introduced in Glover [1986], based on earlier ideas formulated in Glover [1977].[7] A description of the method and its concepts can be found in Glover and Laguna [1997]. TS explicitly uses the history of the search, both to escape from local minima and to implement an explorative strategy. We will first describe a simple version of TS, to introduce the basic concepts. Then, we will explain a more applicable algorithm and finally we will discuss some improvements.

The simple TS algorithm (see Figure 3) applies a best improvement local search as basic ingredient and uses a *short term memory* to escape from local minima and

[7] Related ideas were labelled steepest ascent/mildest descent method in Hansen [1986].

to avoid cycles. The short term memory is implemented as a *tabu list* that keeps track of the most recently visited solutions and forbids moves toward them. The neighborhood of the current solution is thus restricted to the solutions that do not belong to the tabu list. In the following we will refer to this set as *allowed set*. At each iteration the best solution from the allowed set is chosen as the new current solution. Additionally, this solution is added to the tabu list and one of the solutions that were already in the tabu list is removed (usually in a FIFO order). Due to this dynamic restriction of allowed solutions in a neighborhood, TS can be considered as a dynamic neighborhood search technique [Stützle 1999b]. The algorithm stops when a termination condition is met. It might also terminate if the allowed set is empty, that is, if all the solutions in $\mathcal{N}(s)$ are forbidden by the tabu list.[8]

The use of a tabu list prevents from returning to recently visited solutions, therefore it prevents from endless cycling[9] and forces the search to accept even uphill moves. The length l of the tabu list (i.e., the *tabu tenure*) controls the memory of the search process. With small tabu tenures the search will concentrate on small areas of the search space. On the opposite, a large tabu tenure forces the search process to explore larger regions, because it forbids revisiting a higher number of solutions. The tabu tenure can be varied during the search, leading to more robust algorithms. An example can be found in Taillard [1991], where the tabu tenure is periodically reinitialized at random from the interval $[l_{\min}, l_{\max}]$. A more advanced use of a dynamic tabu tenure is presented in Battiti and Tecchiolli [1994] and Battiti and Protasi [1997], where the tabu tenure is increased if there is evidence for repetitions of solutions (thus a higher diversification is needed), while it is decreased if there are no improvements (thus intensification should be boosted). More advanced ways to create dynamic tabu tenure are described in Glover [1990].

However, the implementation of short term memory as a list that contains complete solutions is not practical, because managing a list of solutions is highly inefficient. Therefore, instead of the solutions themselves, solution *attributes* are stored.[10] Attributes are usually components of solutions, moves, or differences between two solutions. Since more than one attribute can be considered, a tabu list is introduced for each of them. The set of attributes and the corresponding tabu lists define the *tabu conditions* which are used to filter the neighborhood of a solution and generate the allowed set. Storing attributes instead of complete solutions is much more efficient, but it introduces a loss of information, as forbidding an attribute means assigning the tabu status to probably more than one solution. Thus, it is possible that unvisited solutions of good quality are excluded from the allowed set. To overcome this problem, *aspiration criteria* are defined which allow to include a solution in the allowed set even if it is forbidden by tabu conditions. Aspiration criteria define the aspiration conditions that are used to construct the allowed set. The most commonly used aspiration criterion selects solutions which are better than the current best one. The complete algorithm, as described above, is reported in Figure 4.

Tabu lists are only one of the possible ways of taking advantage of the history of the search. They are usually identified with the usage of short term memory. Information collected during the whole search process can also be very useful, especially for a strategic guidance of the algorithm. This kind of long-term memory is usually added to TS by referring to four principles: *recency*, *frequency*, *quality* and *influence*. Recency-based memory records for each solution (or attribute)

[8] Strategies for avoiding to stop the search when the allowed set is empty include the choice of the least recently visited solution, even if it is tabu.

[9] Cycles of higher period are possible, since the tabu list has a finite length l which is smaller than the cardinality of the search space.

[10] In addition to storing attributes, some longer term TS strategies also keep complete solutions (e.g., elite solutions) in the memory.

```
s ← GenerateInitialSolution()
InitializeTabuLists(TL_1, ..., TL_r)
k ← 0
while termination conditions not met do
  AllowedSet(s, k) ← {s' ∈ N(s) | s does not violate a tabu condition,
                                  or it satisfies at least one aspiration condition}
  s ← ChooseBestOf(AllowedSet(s, k))
  UpdateTabuListsAndAspirationConditions()
  k ← k + 1
endwhile
```

Fig. 4. Algorithm: Tabu Search (TS).

```
while termination conditions not met do
  s ← ConstructGreedyRandomizedSolution()        % see Figure 6
  ApplyLocalSearch(s)
  MemorizeBestFoundSolution()
endwhile
```

Fig. 5. Algorithm: Greedy Randomized Adaptive Search Procedure (GRASP).

the most recent iteration it was involved in. Orthogonally, frequency-based memory keeps track of how many times each solution (attribute) has been visited. This information identifies the regions (or the subsets) of the solution space where the search was confined, or where it stayed for a high number of iterations. This kind of information about the past is usually exploited to diversify the search. The third principle (i.e., quality) refers to the accumulation and extraction of information from the search history in order to identify good solution components. This information can be usefully integrated in the solution construction. Other metaheuristics (e.g., Ant Colony Optimization) explicitly use this principle to learn about good combinations of solution components. Finally, influence is a property regarding choices made during the search and can be used to indicate which choices have shown to be the most critical. In general, the TS field is a rich source of ideas. Many of these ideas and strategies have been and are currently adopted by other metaheuristics.

TS has been applied to most CO problems; examples for successful applications are the Robust Tabu Search to the QAP [Taillard 1991], the Reactive Tabu Search to the MAXSAT problem [Battiti and Protasi 1997], and to assignment problems [Dell'Amico et al. 1999]. TS approaches dominate the Job Shop Scheduling (JSS) problem area (see, e.g., Nowicki and Smutnicki [1996]) and the Vehicle Routing (VR) area [Gendreau et al. 2001]. Further current applications can be found at [Tabu Search website 2003].

3.4. Explorative Local Search Methods

In this section, we present more recently proposed trajectory methods. These are the Greedy Randomized Adaptive Search Procedure (GRASP), Variable Neighborhood Search (VNS), Guided Local Search (GLS) and Iterated Local Search (ILS).

3.4.1. GRASP. The Greedy Randomized Adaptive Search Procedure (GRASP), see Feo and Resende [1995] and Pitsoulis and Resende [2002], is a simple metaheuristic that combines constructive heuristics and local search. Its structure is sketched in Figure 5. GRASP is an iterative procedure, composed of two phases: solution construction and solution improvement. The best found solution is returned upon termination of the search process.

The solution construction mechanism (see Figure 6) is characterized by two main ingredients: a dynamic constructive heuristic and randomization. Assuming that a solution s consists of a subset of a set of elements (solution components),

```
s ← ∅          % s denotes a partial solution in this case
α ← DetermineCandidateListLength()          % definition of the RCL length
while solution not complete do
  RCL_α ← GenerateRestrictedCandidateList(s)
  x ← SelectElementAtRandom(RCL_α)
  s ← s ∪ {x}
  UpdateGreedyFunction(s)          % update of the heuristic values (see text)
endwhile
```

Fig. 6. Greedy randomized solution construction.

the solution is constructed step-by-step by adding one new element at a time. The choice of the next element is done by picking it uniformly at random from a candidate list. The elements are ranked by means of a heuristic criterion that gives them a score as a function of the (myopic) benefit if inserted in the current partial solution. The candidate list, called *restricted candidate list* (RCL), is composed of the best α elements. The heuristic values are updated at each step, thus the scores of elements change during the construction phase, depending on the possible choices. This constructive heuristic is called *dynamic*, in contrast to the *static* one which assigns a score to elements only before starting the construction. For instance, one of the static heuristics for the TSP is based on arc costs: the lower the cost of an arc, the higher its score. An example of a dynamic heuristic is the *cheapest insertion* heuristic, where the score of an element is evaluated depending on the current partial solution.

The length α of the restricted candidate list determines the strength of the heuristic bias. In the extreme case of $\alpha = 1$ the best element would be added, thus the construction would be equivalent to a deterministic Greedy Heuristic. On the opposite, in case $\alpha = n$ the construction would be completely random (indeed, the choice of an element from the candidate list is done at random). Therefore, α is a critical parameter which influences the sampling of the search space. In Pitsoulis and Resende [2002] the most important schemes to define α are listed. The simplest scheme is, trivially, to keep α constant; it can also be changed at each iteration, either randomly or by means of an adaptive scheme.

The second phase of the algorithm is a local search process, which may be a basic local search algorithm such as iterative improvement, or a more advanced technique such as SA or TS. GRASP can be effective if two conditions are satisfied:

—the solution construction mechanism samples the most promising regions of the search space;

—the solutions constructed by the constructive heuristic belong to basins of attraction of different locally minimal solutions;

The first condition can be met by the choice of an effective constructive heuristic and an appropriate length of the candidate list, whereas the second condition can be met by choosing the constructive heuristic and the local search in a way such that they fit well.

The description of GRASP as given above indicates that a basic GRASP does not use the history of the search process.[11] The only memory requirement is for storing the problem instance and for keeping the best so-far solution. This is one of the reasons why GRASP is often outperformed by other metaheuristics. However, due to its simplicity, it is generally very fast and it is able to produce quite good solutions in a very short amount of computation time. Furthermore, it can be successfully integrated into other search techniques. Among the applications of GRASP, we mention the JSS problem

[11]However, some extensions in this direction are cited in Pitsoulis and Resende [2002], and an example for a metaheuristic method using an adaptive greedy procedure depending on search history is Squeaky Wheel Optimization (SWO) [Joslin and Clements 1999].

```
Select a set of neighborhood structures N_k, k = 1, ..., k_max
s ← GenerateInitialSolution()
while termination conditions not met do
  k ← 1
  while k < k_max do                    % Inner loop
    s' ← PickAtRandom(N_k(s))           % Shaking phase
    s'' ← LocalSearch(s')
    if (f(s'') < f(s)) then
      s ← s''
      k ← 1
    else
      k ← k + 1
    endif
  endwhile
endwhile
```

Fig. 7. Algorithm: Variable Neighborhood Search (VNS).

[Binato et al. 2001], the graph planarization problem [Resende and Ribeiro 1997] and assignment problems [Prais and Ribeiro 2000]. A detailed and annotated bibliography references many more applications [Festa and Resende 2002].

3.4.2. Variable Neighborhood Search. Variable Neighborhood Search (VNS) is a metaheuristic proposed in Hansen and Mladenović [1999, 2001], which explicitly applies a strategy based on dynamically changing neighborhood structures. The algorithm is very general and many degrees of freedom exist for designing variants and particular instantiations.[12]

At the initialization step, a set of neighborhood structures has to be defined. These neighborhoods can be arbitrarily chosen, but often a sequence $|\mathcal{N}_1| < |\mathcal{N}_2| < \cdots < |\mathcal{N}_{k_{max}}|$ of neighborhoods with increasing cardinality is defined.[13] Then an initial solution is generated, the neighborhood index is initialized and the algorithm iterates until a stopping condition is met (see Figure 7). VNS' main cycle is composed of three phases: *shaking*, *local search* and *move*. In the *shaking* phase a solution s' in the kth neighborhood of the current solution s is randomly selected. Then, s' becomes the local search starting point. The local search can use any neighborhood structure and is not restricted to the set of neighborhood structures $\mathcal{N}_k$, $k = 1, \ldots, k_{\max}$. At the end of the local search process (terminated as soon as a predefined termination condition is verified) the new solution s'' is compared with s and, if it is better, it replaces s and the algorithm starts again with $k = 1$. Otherwise, k is incremented and a new shaking phase starts using a different neighborhood.

The objective of the shaking phase is to perturb the solution so as to provide a good starting point for the local search. The starting point should belong to the basin of attraction of a different local minimum than the current one, but should not be "too far" from s, otherwise the algorithm would degenerate into a simple random multi-start. Moreover, choosing s' in the neighborhood of the current best solution is likely to produce a solution that maintains some good features of the current one.

The process of changing neighborhoods in case of no improvements corresponds to a diversification of the search. In particular the choice of neighborhoods of increasing cardinality yields a progressive diversification. The effectiveness of this dynamic neighborhood strategy can be explained by the fact that a "bad" place on the search landscape given by one neighborhood could be a "good" place on the search landscape given by another neighborhood.[14] Moreover, a solution that is locally optimal with respect to a neighborhood is probably not locally optimal with respect to another neighborhood. These concepts are known as "*One Operator, One Landscape*" and explained in Jones [1995a, 1995b]. The core idea is that the neighborhood structure determines the topological properties of the search landscape, that is, each neighborhood defines one landscape. The properties of a landscape are in general different from those of other landscapes, therefore

[12] The variants described in the following are also described in Hansen and Mladenović [1999, 2001].

[13] In principle they could be one included in the other, $\mathcal{N}_1 \subset \mathcal{N}_2 \subset \cdots \subset \mathcal{N}_{k_{max}}$. Nevertheless, such a sequence might produce an inefficient search, because a large number of solutions could be revisited.

[14] A "good" place in the search space is an area from which a good local minimum can be reached.

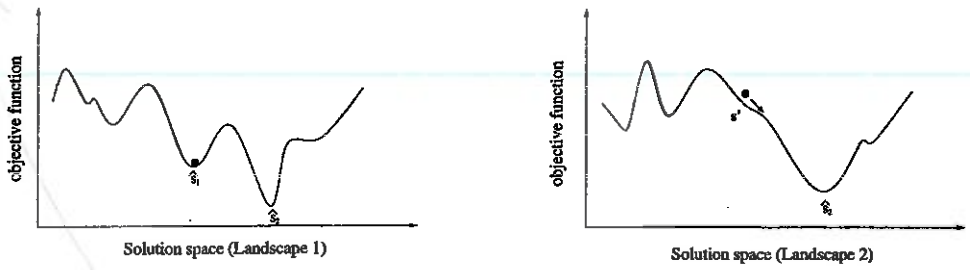

Fig. 8. Two search landscapes defined by two different neighborhoods. On the landscape that is shown in the graphic on the left, the best improvement local search stops at $\hat{s}_1$, while it proceeds till a better local minimum $\hat{s}_2$ on the landscape that is shown in the graphic on the right.

a search strategy performs differently on them (see an example in Figure 8).

This property is directly exploited by a local search called Variable Neighborhood Descent (VND). In VND a *best improvement* local search (see Section 3.1) is applied, and, in case a local minimum is found, the search proceeds with another neighborhood structure. The VND algorithm can be obtained by substituting the inner loop of the VNS algorithm (see Figure 7) with the following pseudo-code:

```
s' ← ChooseBestOf(N_k(s))
if (f(s') < f(s))
then    % i.e., if a better solution is found in N_k(s)
  s ← s'
else          % i.e., s is a local minimum
  k ← k + 1
endif
```

As can be observed from the description as given above, the choice of the neighborhood structures is the critical point of VNS and VND. The neighborhoods chosen should exploit different properties and characteristics of the search space, that is, the neighborhood structures should provide different *abstractions* of the search space. A variant of VNS is obtained by selecting the neighborhoods in such a way as to produce a problem decomposition (the algorithm is called Variable Neighborhood Decomposition Search—VNDS). VNDS follows the usual VNS scheme, but the neighborhood structures and the local search are defined on sub-problems. For each solution, all attributes (usually variables) are kept fixed except for k of them. For each, k, a neighborhood structure N_k is defined. Local search only regards changes on the variables belonging to the sub-problem it is applied to. The inner loop of VNDS is the following:

```
s' ← PickAtRandom(N_k(s))      % s and s' differ in k attributes
s'' ← LocalSearch(s',Attributes)  % only moves involving the k attributes are allowed
if (f(s'') < f(s)) then
  s ← s''
  k ← 1
else
  k ← k + 1
endif
```

The decision whether to perform a move can be varied as well. The acceptance criterion based on improvements is strongly steepest descent-oriented and it might not be suited to effectively explore the search space. For example, when local minima are clustered, VNS can quickly find the best optimum in a cluster, but it has no guidance to leave that cluster and find another one. Skewed VNS (SVNS) extends VNS by providing a more flexible acceptance criterion that takes also into account the distance from the current solution.[15] The new acceptance criterion is the following: besides always accepting improvements, worse solutions can be accepted if the distance from the current one is less than a value $\alpha\rho(s, s'')$. The function $\rho(s, s'')$ measures the distance between s and s''

[15] A distance measure between solutions has thus to be formally defined.

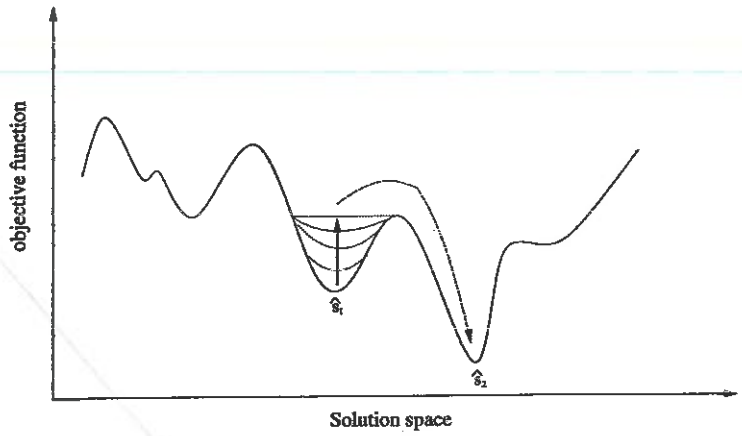

Fig. 9. Basic GLS idea: Escaping from a valley in the landscape by increasing the objective function value of its solutions.

and α is a parameter that weights the importance of the distance between the two solutions in the acceptance criterion. The inner loop of SVNS can be sketched as follows:

if $(f(s'') - \alpha\rho(s, s'') < f(s))$ **then**
  $s \leftarrow s''$
  $k \leftarrow 1$
else
  $k \leftarrow k + 1$
endif

VNS and its variants have been successfully applied to graph based CO problems such as the p-Median problem [Hansen and Mladenović 1997], the degree constrained minimum spanning tree problem [Ribeiro and Souza 2002], the Steiner tree problem [Wade and Rayward-Smith 1997] and the k-Cardinality Tree (KCT) problem [Mladenović and Urošević 2001]. References to more applications can be found in Hansen and Mladenović [2001].

3.4.3. Guided Local Search. Tabu Search and Variable Neighborhood Search explicitly deal with dynamic neighborhoods with the aim of efficiently and effectively exploring the search space. A different approach for guiding the search is to dynamically change the objective function. Among the most general methods that use this approach is Guided Local Search (GLS) [Voudouris and Tsang 1999; Voudouris 1997].

The basic GLS principle is to help the search to gradually move away from local minima by changing the search landscape. In GLS, the set of solutions and the neighborhood structure are kept fixed, while the objective function f is dynamically changed with the aim of making the current local optimum "less desirable". A pictorial description of this idea is given in Figure 9.

The mechanism used by GLS is based on *solution features*, which may be any kind of properties or characteristics that can be used to discriminate between solutions. For example, solution features in the TSP could be arcs between pairs of cities, while in the MAXSAT problem they could be the number of unsatisfied clauses. An indicator function $I_i(s)$ indicates whether the feature i is present in solution s:

$$I_i(s) = \begin{cases} 1 & : \text{ if feature } i \text{ is present in} \\ & : \quad \text{solution } s \\ 0 & : \text{ otherwise.} \end{cases}$$

The objective function f is modified to yield a new objective function f' by adding a term that depends on the m features:

$$f'(s) = f(s) + \lambda \sum_{i=1}^{m} p_i \cdot I_i(s),$$

where p_i are called *penalty parameters* and λ is called the *regularization*

```
s ← GenerateInitialSolution()
while termination conditions not met do
  s ← LocalSearch(s, f')
  for all feature i with maximum utility Util(s, i) do
    p_i ← p_i + 1
  endfor
  Update(f', p)            % p is the penalty vector
endwhile
```

Fig. 10. Algorithm: Guided Local Search (GLS).

parameter. The penalty parameters weight the importance of the features: the higher p_i, the higher the importance of feature i, thus the higher the cost of having that feature in the solution. The regularization parameter balances the relevance of features with respect to the original objective function.

The algorithm (see Figure 10) works as follows: It starts from an initial solution and applies a local search method until a local minimum is reached. Then the array $\mathbf{p} = (p_1, \ldots, p_m)$ of penalties is updated by incrementing some of the penalties and the local search is started again. The penalized features are those that have the maximum *utility*:

$$Util(s, i) = I_i(s) \cdot \frac{c_i}{1 + p_i},$$

where c_i are *costs* assigned to every feature i giving a heuristic evaluation of the relative importance of features with respect to others. The higher the cost, the higher the utility of features. Nevertheless, the cost is scaled by the penalty parameter to prevent the algorithm from being totally biased toward the cost and to make it sensitive to the search history.

The penalties update procedure can be modified by adding a multiplicative rule to the simple incrementing rule (that is applied at each iteration). The multiplicative rule has the form: $p_i \leftarrow p_i \cdot \alpha$, where $\alpha \in (0, 1)$. This rule is applied with a lower frequency than the incrementing one (for example every few hundreds of iterations) with the aim of smoothing the weights of penalized features so as to prevent the landscape from becoming too rugged. It is important to note that the penalties update rules are often very sensitive to the problem instance.

GLS has been successfully applied to the weighted MAXSAT [Mills and Tsang 2000], the VR problem [Kilby et al. 1999], the TSP and the QAP [Voudouris and Tsang 1999].

3.4.4. Iterated Local Search. We conclude this presentation of explorative strategies with Iterated Local Search (ILS), the most general scheme among the explorative strategies. On the one hand, its generality makes it a framework for other metaheuristics (such as VNS); on the other hand, other metaheuristics can be easily incorporated as subcomponents. ILS is a simple but powerful metaheuristic algorithm [Stützle 1999a,1999b; Lourenço et al. 2001,2002; Martin et al. 1991]. It applies local search to an initial solution until it finds a local optimum; then it perturbs the solution and it restarts local search. The importance of the perturbation is obvious: too small a perturbation might not enable the system to escape from the basin of attraction of the local optimum just found. On the other side, too strong a perturbation would make the algorithm similar to a random restart local search.

A local search is effective if it is able to find good local minima, that is, if it can find the basin of attraction of those states. When the search space is wide and/or when the basins of attraction of good local optima are small,[16] a simple multi-start algorithm is almost useless. An effective search could be designed as a trajectory only in the set of local optima $\hat{S}$, instead of in the set S of all the states. Unfortunately, in most cases there is no feasible way of introducing a neighborhood structure for $\hat{S}$. Therefore, a trajectory along local optima $\hat{s_1}, \hat{s_2}, \ldots, \hat{s_t}$ is performed, without explicitly introducing a neighborhood structure, by applying the following scheme:

[16]The basin of attraction size of a point s (in a finite space), is defined as the fraction of initial states of trajectories which converge to point s.

```
s0 ← GenerateInitialSolution()
ŝ ← LocalSearch(s0)
while termination conditions not met do
  s' ← Perturbation(ŝ, history)
  ŝ' ← LocalSearch(s')
  ŝ ← ApplyAcceptanceCriterion(ŝ, ŝ', history)
endwhile
```

Fig. 11. Algorithm: Iterated Local Search (ILS).

(1) Execute local search (LS) from an initial state s until a local minimum $\hat{s}$ is found.
(2) *Perturb* $\hat{s}$ and obtain s'.
(3) Execute LS from s' until a local minimum $\hat{s}'$ is reached.
(4) On the basis of an *acceptance criterion* decide whether to set $\hat{s} \leftarrow \hat{s}'$.
(5) Goto step 2.

The requirement on the *perturbation* of $\hat{s}$ is to produce a starting point for local search such that a local minimum different from $\hat{s}$ is reached. However, this new local minimum should be *closer* to $\hat{s}$ than a local minimum produced by a random restart. The *acceptance criterion* acts as a counterbalance, as it filters and gives feedback to the perturbation action, depending on the characteristics of the new local minimum. A high level description of ILS as it is described in Lourenço et al. [2002] is given in Figure 11. Figure 12 shows a possible (lucky) ILS step.

The design of ILS algorithms has several degrees of freedom in the choice of the initial solution, perturbation and acceptance criteria. A key role is played by the *history* of the search which can be exploited both in form of short and long term memory.

The construction of initial solutions should be fast (computationally not expensive), and initial solutions should be a good starting point for local search. The fastest way of producing an initial solution is to generate it at random; however, this is the easiest way for problems that are constrained, whilst in other cases the construction of a feasible solution requires also constraint checking. Constructive methods, guided by heuristics, can also be adopted. It is worth underlining that an initial solution is considered a good starting point depending on the particular LS applied and on the problem structure, thus the algorithm designer's goal is to find a trade-off between speed and quality of solutions.

The perturbation is usually nondeterministic in order to avoid cycling. Its most important characteristic is the *strength*, roughly defined as the amount of changes made on the current solution. The strength can be either fixed or variable. In the first case, the distance between $\hat{s}$ and s' is kept constant, independently of the problem size. However, a variable strength is in general more effective, since it has been experimentally found that, in most of the problems, the bigger the problem size, the larger should be the strength. More sophisticated schemes are possible; for example, the strength can be adaptive: it increases when more diversification is needed and it decreases when intensification seems preferable. VNS and its variants belong to this category. A second choice is the mechanism to perform perturbations. This may be a random mechanism, or the perturbation may be produced by a (semi-)deterministic method (e.g., a LS different from the one used in the main algorithm).

The third important component is the acceptance criterion. Two extreme examples consist in (1) accepting the new local optimum only in case of improvement and (2) in always accepting the new solution. In-between, there are several possibilities. For example, it is possible to adopt a kind of annealing schedule: accept all the improving new local optima and accept also the nonimproving ones with a probability that is a function of the temperature T and the difference of objective function values. In formulas:

$$\mathbf{p}(Accept(\hat{s}, \hat{s}', history)) = \begin{cases} 1 & \text{if } f(\hat{s}') < f(\hat{s}) \\ \exp\left(-\dfrac{f(\hat{s}') - f(\hat{s})}{T}\right) & \text{otherwise} \end{cases}$$

The cooling schedule can be either monotonic (non-increasing in time) or

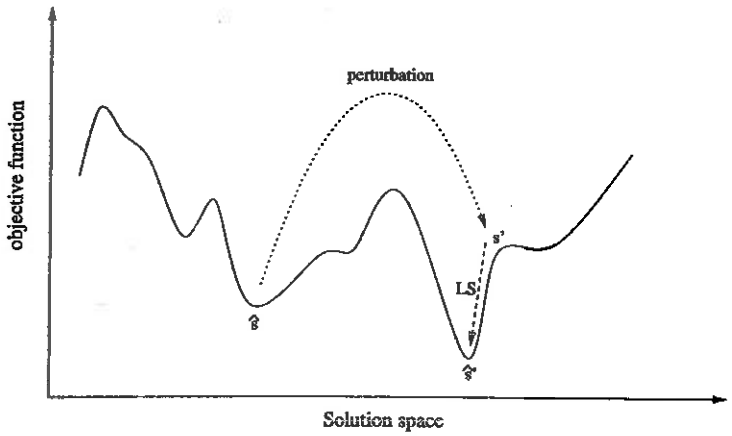

Fig. 12. A desirable ILS step: the local minimum $\hat{s}$ is perturbed, then LS is applied and a new local minimum is found.

non-monotonic (adapted to tune the balance between diversification and intensification). The nonmonotonic schedule is particularly effective if it exploits the history of the search, in a way similar to the Reactive Tabu Search [Taillard 1991] mentioned at the end of the section about Tabu Search. When intensification seems no longer effective, a diversification phase is needed and the temperature is increased.

Examples for successful applications of ILS are to the TSP [Martin and Otto 1996; Johnson and McGeoch 1997], to the QAP [Lourenço et al. 2002], and to the Single Machine Total Weighted Tardiness (SMTWT) problem [den Besten et al. 2001]. References to other applications can be found in Lourenço et al. [2002].

4. POPULATION-BASED METHODS

Population-based methods deal in every iteration of the algorithm with a set (i.e., a population) of solutions[17] rather than with a single solution. As they deal with a population of solutions, population-based algorithms provide a natural, intrinsic way for the exploration of the search space. Yet, the final performance depends strongly on the way the population is manipulated. The most studied population-based methods in combinatorial optimization are Evolutionary Computation (EC) and Ant Colony Optimization (ACO). In EC algorithms, a population of individuals is modified by recombination and mutation operators, and in ACO a colony of artificial ants is used to construct solutions guided by the pheromone trails and heuristic information.

[17]In general, especially in EC algorithms, we talk about a population of individuals rather than solutions.

4.1. Evolutionary Computation

Evolutionary Computation (EC) algorithms are inspired by nature's capability to evolve living beings well adapted to their environment. EC algorithms can be succinctly characterized as computational models of evolutionary processes. At each iteration a number of operators is applied to the individuals of the current population to generate the individuals of the population of the next generation (iteration). Usually, EC algorithms use operators called *recombination* or *crossover* to recombine two or more individuals to produce new individuals. They also use *mutation* or *modification* operators which cause a self-adaptation of individuals. The driving force in evolutionary algorithms is the *selection* of individuals based on their *fitness* (this can be the value of an objective function or the result of a simulation experiment, or some other kind of quality measure). Individuals with a higher

GA = Genetic Algorithm

fitness have a higher probability to be chosen as members of the population of the next iteration (or as parents for the generation of new individuals). This corresponds to the principle of *survival of the fittest* in natural evolution. It is the capability of nature to adapt itself to a changing environment, which gave the inspiration for EC algorithms.

There has been a variety of slightly different EC algorithms proposed over the years. Basically they fall into three different categories which have been developed independently from each other. These are Evolutionary Programming (EP) developed by Fogel [1962] and Fogel et al. [1966], Evolutionary Strategies (ES) proposed by Rechenberg [1973] and Genetic Algorithms initiated by Holland [1975] (see Goldberg [1989], Mitchell [1998], Reeves and Rowe [2002], and Vose [1999] for further references). EP arose from the desire to generate machine intelligence. While EP originally was proposed to operate on discrete representations of finite state machines, most of the present variants are used for continuous optimization problems. The latter also holds for most present variants of ES, whereas GAs are mainly applied to solve combinatorial optimization problems. Over the years, there have been quite a few overviews and surveys about EC methods. Among those are the ones by Bäck [1996], Fogel [1994], Spears et al. [1993] and Michalewicz and Michalewicz [1997]. Calégary et al. [1999] propose a taxonomy of EC algorithms.

In the following we provide a "combinatorial optimization"-oriented introduction to EC algorithms. For doing this, we follow an overview work by Hertz and Kobler [2000], which gives, in our opinion, a good overview of the different components of EC algorithms and of the possibilities to define them.

Figure 13 shows the basic structure of every EC algorithm. In this algorithm, P denotes the population of individuals. A population of offspring is generated by the application of *recombination* and *mutation* operators and the individuals for the next population are *selected* from the union of the old population and the offspring population. The main features of an EC algorithm are outlined in the following.

```
P ← GenerateInitialPopulation()
Evaluate(P)
while termination conditions not met do
  P' ← Recombine(P)
  P'' ← Mutate(P')
  Evaluate(P'')
  P ← Select(P'' ∪ P)
endwhile
```

Fig. 13. Algorithm: Evolutionary Computation (EC).

Description of the Individuals. EC algorithms handle populations of individuals. These individuals are not necessarily solutions of the considered problem. They may be partial solutions, or sets of solutions, or any object which can be transformed into one or more solutions in a structured way. Most commonly used in combinatorial optimization is the representation of solutions as bit-strings or as permutations of n integer numbers. Tree-structures or other complex structures are also possible. In the context of Genetic Algorithms, individuals are called *genotypes*, whereas the solutions that are encoded by individuals are called *phenotypes*. This is to differentiate between the representation of solutions and solutions themselves. The choice of an appropriate representation is crucial for the success of an EC algorithm. Holland's [1975] schema analysis and Radcliffe's [1991] generalization to formae are examples of how theory can help to guide representation choices.

Evolution Process. In each iteration, it has to be decided which individuals will enter the population of the next iteration. This is done by a selection scheme. To choose, the individuals for the next population exclusively from the offspring is called *generational replacement*. If it is possible to transfer individuals of the current population into the next population, then we deal with a so-called *steady state* evolution process.

Most EC algorithms work with populations of fixed size keeping at least the best individual always in the current

population. It is also possible to have a variable population size. In case of a continuously shrinking population size, the situation where only one individual is left in the population (or no crossover partners can be found for any member of the population) might be one of the stopping conditions of the algorithm.

Neighborhood Structure. A neighborhood function $\mathcal{N}_{EC} : \mathcal{I} \rightarrow 2^{\mathcal{I}}$ on the set of individuals $\mathcal{I}$ assigns to every individual $i \in \mathcal{I}$ a set of individuals $\mathcal{N}_{EC}(i) \subseteq \mathcal{I}$ which are permitted to act as recombination partners for i to create offspring. If an individual can be recombined with any other individual (as for example in the simple GA) we talk about *unstructured* populations, otherwise we talk about *structured* populations. An example for an EC algorithm that works on structured populations is the Parallel Genetic Algorithm proposed by Mühlenbein [1991].

Information Sources. The most common form of information sources to create offspring (i.e., new individuals) is a couple of parents (two-parent crossover). But there are also recombination operators that operate on more than two individuals to create a new individual (multi-parent crossover), see Eiben et al. [1994]. More recent developments even use population statistics for generating the individuals of the next population. Examples are the recombination operators called Gene Pool Recombination [Mühlenbein and Voigt 1995] and Bit-Simulated Crossover [Syswerda 1993] which make use of a distribution over the search space given by the current population to generate the next population.

Infeasibility. An important characteristic of an EC algorithm is the way it deals with infeasible individuals. When recombining individuals, the offspring might be potentially infeasible. There are basically three different ways to handle such a situation. The most simple action is to *reject* infeasible individuals. Nevertheless, for many problems (e.g., for timetabling problems) it might be very difficult to find feasible individuals. Therefore, the strategy of *penalizing* infeasible individuals in the function that measures the quality of an individual is sometimes more appropriate (or even unavoidable). The third possibility consists in trying to *repair* an infeasible solution (see Eiben and Ruttkay [1997] for an example).

Intensification Strategy. In many applications it proved to be quite beneficial to use improvement mechanisms to improve the fitness of individuals. EC algorithms that apply a local search algorithm to every individual of a population are often called Memetic Algorithms [Moscato 1989, 1999]. While the use of a population ensures an exploration of the search space, the use of local search techniques helps to quickly identify "good" areas in the search space.

Another intensification strategy is the use of recombination operators that explicitly try to combine "good" parts of individuals (rather than, e.g., a simple one-point crossover for bit-strings). This may guide the search performed by EC algorithms to areas of individuals with certain "good" properties. Techniques of this kind are sometimes called *linkage learning* or *building block learning* (see Goldberg et al. [1991], van Kemenade [1996], Watson et al. [1998], and Harik [1999] as examples). Moreover, generalized recombination operators have been proposed in the literature, which incorporate the notion of "neighborhood search" into EC. An example can be found in Rayward-Smith [1994].

Diversification Strategy. One of the major difficulties of EC algorithms (especially when applying local search) is the premature convergence toward sub-optimal solutions. The most simple mechanism to diversify the search process is the use of a mutation operator. The simple form of a mutation operator just performs a small random perturbation of an individual, introducing a kind of *noise*. In order to avoid premature convergence there are ways of maintaining the population diversity. Probably the oldest strategies are *crowding* [DeJong 1975] and its close relative, *preselection*. Newer strategies are *fitness sharing* [Goldberg and Richardson 1987], respectively *niching*, whereby the reproductive fitness allocated to an individual

```
Initial Phase:
SeedGeneration()
repeat
  DiversificationGenerator()
  Improvement()
  ReferenceSetUpdate()
until the reference set is of cardinality n

Scatter Search/Path Relinking Phase:
repeat
  SubsetGeneration()
  SolutionCombination()
  Improvement()
  ReferenceSetUpdate()
until termination criteria met
```

Fig. 14. Algorithm: Scatter Search and Path Relinking.

in a population is reduced proportionally to the number of other individuals that share the same region of the search space.

This concludes the list of the main features of EC algorithms. EC algorithms have been applied to most CO problems and optimization problems in general. Recent successes were obtained in the rapidly growing bioinformatics area (see, e.g., Fogel et al. [2002]), but also in multiobjective optimization [Coello Coello 2000], and in evolvable hardware [Sipper et al. 1997]. For an extensive collection of references to EC applications, we refer to Bäck et al. [1997]. In the following two sections, we are going to introduce two other populations-based methods that are sometimes also regarded as being EC algorithms.

4.1.1. Scatter Search and Path Relinking. Scatter Search and its generalized form called Path Relinking [Glover 1999; Glover et al. 2000] differ from EC algorithms mainly by providing unifying principles for joining (or recombining) solutions based on generalized path constructions in Euclidean or neighborhood spaces. They also incorporate some ideas originating from Tabu Search methods, as, for example, the use of adaptive memory and associated memory-exploiting mechanisms. The template for Scatter Search (respectively, Path Relinking) is shown in Figure 14.

Scatter Search (respectively, Path Relinking) is a search strategy that generates a set of solutions from a chosen set of *reference solutions* corresponding to feasible solutions to the problem under consideration. This is done by making combinations of subsets of the current set of reference solutions. The resulting solutions are called *trial solutions*. These trial solutions may be infeasible solutions and are therefore usually modified by means of a repair procedure that transforms them into feasible solutions. An improvement mechanism is then applied in order to try to improve the set of trial solutions (usually this improvement procedure is a local search). These improved solutions form the set of *dispersed solutions*. The new set of reference solutions that will be used in the next iteration is selected from the current set of reference solutions and the newly created set of dispersed solutions. The components of the pseudo-code, which is shown in Figure 14, are explained in the following:

SeedGeneration(): One or more seed solutions, which are arbitrary trial solutions, are created and used to initiate the remainder of the method.

DiversificationGenerator(): This is a procedure to generate a collection of diverse trial solutions from an arbitrary trial solution (or seed solution).

Improvement(): In this procedure, an improvement mechanism—usually a local search—is used to transform a trial solution into one or more enhanced trial solutions. Neither the input nor the output solutions are required to be feasible, though the output solutions will more usually be expected to be so. It might be necessary to apply repair methods to infeasible solutions.

ReferenceSetUpdate(): The procedure for updating the reference set is responsible for building and maintaining a reference set consisting of a number of "best" solutions found in the course of the algorithm. The attribute "best" covers features such as quality of solutions and diversity of solutions (the solutions in the reference set should be of good quality and they should be diverse).

SubsetGeneration(): This method operates on the reference set, to produce a subset of its solutions as a basis for creating combined solutions.

SolutionCombination(): A procedure to transform a given subset of solutions produced by the subset generation method into one or more combined solutions. In Scatter Search, which was introduced for solutions encoded as points in the Euclidean space, new solutions are created by building linear combinations of reference solutions using both positive and negative weights. This means that trial solutions can be both, inside and outside the convex region spanned by the reference solutions. In Path Relinking, the concept of combining solutions by making linear combinations of reference points is generalized to neighborhood spaces. Linear combinations of points in the Euclidean space can be re-interpreted as paths between and beyond solutions in a neighborhood space. To generate the desired paths, it is only necessary to select moves that satisfy the following condition: upon starting from an *initiating solution*, the moves must progressively introduce attributes contributed by a *guiding solution*. Multiparent path generation possibilities emerge in Path Relinking by considering the combined attributes provided by a set of guiding solutions, where these attributes are weighted to determine which moves are given higher priority.

Scatter Search enjoys increasing interest in recent years. Among other problems, it has been applied to multiobjective assignment problems [Laguna et al. 2000] and the Linear Ordering Problem (LOP) [Campos et al. 2001]. For further references, we refer to [Glover et al. 2002]. Path relinking is often used as a component in metaheuristics such as Tabu Search [Laguna et al. 1999] and GRASP [Aiex et al. 2003; Laguna and Martí 1999].

4.1.2. Estimation of Distribution Algorithms. In the last decade, more and more researchers tried to overcome the drawbacks of usual recombination operators of EC algorithms, which are likely to break good building blocks. So, a number of algorithms—sometimes called Estimation of Distribution Algorithms (EDA) [Mühlenbein and Paaß 1996]—have been developed (see Figure 15 for the algorithmic framework). These algorithms, which have a theoretical foundation in probability theory, are also based on populations that evolve as the search progresses. EDAs use probabilistic modelling of promising solutions to estimate a distribution over the search space, which is then used to produce the next generation by sampling the search space according to the estimated distribution. After every iteration, the distribution is re-estimated. For a survey of EDAs, see [Pelikan et al. 1999b].

```
P ← InitializePopulation()
while termination criteria not met do
  P_sel ← Select(P)                    % P_sel ⊆ P
  p(x) = p(x | P_sel) ← EstimateProbabilityDistribution()
  P ← SampleProbabilityDistribution()
endwhile
```

Fig. 15. Algorithm: Estimation of Distribution Algorithms (EDAs).

One of the first EDAs that was proposed for Combinatorial Optimization is called Population-based Incremental Learning (PBIL) [Baluja 1994; Baluja and Caruana 1995]. The objective of this method is to create a real valued probability vector (each position corresponds to a binary decision variable) which—when used to sample the search space—generates high quality solutions with high probability. Initially, the values of the probability vector are initialized to 0.5 (for each variable there is equal probability to be set to 0 or 1). The goal of shifting the values of this probability vector in order to generate high quality solutions is accomplished as follows: a number of solution vectors are generated according to the probability vector. Then, the probability vector is shifted toward the generated solution vector(s) with highest quality. The distance that the probability vector is shifted depends on the learning rate parameter. Then, a mutation operator is applied to the probability vector. After that, the cycle is repeated. The probability vector can be regarded as a prototype vector for generating high-quality solution vectors

with respect to the available knowledge about the search space. The drawback of this method is the fact that it does not automatically provide a way to deal with constrained problems. In contrast to PBIL, which estimates a distribution of promising solutions assuming that the decision variables are independent, various other approaches try to estimate distributions taking into account dependencies between decision variables. An example for EDAs regarding pairwise dependencies between decision variables is MIMIC [de Bonet et al. 1997] and an example for multivariate dependencies is the Bayesian Optimization Algorithm (BOA) [Pelikan et al. 1999a].

The field of EDAs is still quite young and much of the research effort is focused on methodology rather than high-performance applications. Applications to Knapsack problems, the Job Shop Scheduling (JSS) problem, and other CO problems can be found in Larrañaga and Lozano [2002].

4.2. Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic approach proposed in Dorigo 1992, 1996, 1999. In the course of this section, we keep close to the description of ACO as given in Dorigo and Di Caro [1999]. The inspiring source of ACO is the foraging behavior of real ants. This behavior—as described by Deneubourg et al. [1990]—enables ants to find shortest paths between food sources and their nest. While walking from food sources to the nest and vice versa, ants deposit a substance called *pheromone* on the ground. When they decide about a direction to go, they choose with higher probability paths that are marked by stronger pheromone concentrations. This basic behavior is the basis for a cooperative interaction which leads to the emergence of shortest paths.

ACO algorithms are based on a parametrized probabilistic model—the *pheromone model*—that is used to model the chemical pheromone trails. Artificial ants incrementally construct solutions by adding opportunely defined solution components to a partial solution under consideration.[18] For doing that, artificial ants perform randomized walks on a completely connected graph $\mathcal{G}=(\mathcal{C},\mathcal{L})$ whose vertices are the solution components $\mathcal{C}$ and the set $\mathcal{L}$ are the connections. This graph is commonly called *construction graph*. When a constrained CO problem is considered, the problem constraints Ω are built into the ants' constructive procedure in such a way that in every step of the construction process only feasible solution components can be added to the current partial solution. In most applications, ants are implemented to build feasible solutions, but sometimes it is unavoidable to also let them construct infeasible solutions. Components $c_i \in \mathcal{C}$ can have associated a *pheromone trail parameter* $\mathcal{T}_i$, and connections $l_{ij} \in \mathcal{L}$ can have associated a *pheromone trail parameter* $\mathcal{T}_{ij}$. The set of all pheromone trail parameters is denoted by $\mathcal{T}$. The values of these parameters—the *pheromone values*—are denoted by τ_i, respectively τ_{ij}. Furthermore, components and connections can have associated a *heuristic value* η_i, respectively η_{ij}, representing *a priori* or run time heuristic information about the problem instance. We henceforth denote the set of all heuristic values by $\mathcal{H}$. These values are used by the ants to make probabilistic decisions on how to move on the construction graph. The probabilities involved in moving on the construction graph are commonly called *transition probabilities*. The first ACO algorithm proposed in the literature is called Ant System (AS) [Dorigo et al. 1996]. The pseudo-code for this algorithm is shown in Figure 16. For the sake of simplicity, we restrict the following description of AS to pheromone trail parameters and heuristic information on solution components.

In this algorithm, $\mathcal{A}$ denotes the set of ants and s_a denotes the solution constructed by ant $a \in \mathcal{A}$. After the initialization of the pheromone values, at each step of the algorithm each ant constructs a

[18] Therefore, the ACO metaheuristic can be applied to any CO problem for which a constructive procedure can be defined.

```
InitializePheromoneValues(T)
while termination conditions not met do
  for all ants a ∈ A do
    s_a ← ConstructSolution(T,H)
  endfor
  ApplyOnlineDelayedPheromoneUpdate(T,{s_a | a ∈ A})
endwhile
```

Fig. 16. Algorithm: Ant System (AS).

solution. These solutions are then used to update the pheromone values. The components of this algorithm are explained in more detail in the following.

InitializePheromoneValues($\mathcal{T}$): At the beginning of the algorithm the pheromone values are initialized to the same small value $ph > 0$.

ConstructSolution($\mathcal{T},\mathcal{H}$): In the construction phase an ant incrementally builds a solution by adding solution components to the partial solution constructed so far. The probabilistic choice of the next solution component to be added is done by means of transition probabilities, which in AS are determined by the following *state transition rule*:

$$\mathbf{p}(c_r|s_a[c_l]) = \begin{cases} \dfrac{[\eta_r]^\alpha\,[\tau_r]^\beta}{\sum_{c_u \in J(s_a[c_l])}[\eta_u]^\alpha\,[\tau_u]^\beta} & \text{if } c_r \in J(s_a[c_l]) \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

In this formula, α and β are parameters to adjust the relative importance of heuristic information and pheromone values and $J(s_a[c_l])$ denotes the set of solution components that are allowed to be added to the partial solution $s_a[c_l]$, where c_l is the last component that was added.

ApplyOnlineDelayedPheromoneUpdate ($\mathcal{T},\{s_a \mid a \in \mathcal{A}\}$): Once all ants have constructed a solution, the *online delayed pheromone update rule* is applied:

$$\tau_j \leftarrow (1-\rho)\cdot\tau_j + \sum_{a\in\mathcal{A}} \Delta\tau_j^{s_a} \quad (2)$$

$\forall\, T_j \in \mathcal{T}$, where

$$\Delta\tau_j^{s_a} = \begin{cases} F(s_a) & \text{if } c_j \text{ is a component of } s_a \\ 0 & \text{otherwise,} \end{cases} \quad (3)$$

```
while termination conditions not met do
  ScheduleActivities
    AntBasedSolutionConstruction()
    PheromoneUpdate()
    DaemonActions()          % optional
  end ScheduleActivities
endwhile
```

Fig. 17. Algorithm: Ant Colony Optimization (ACO).

where $F: S \mapsto \mathbb{R}^+$ is a function that satisfies $f(s) < f(s') \Rightarrow F(s) \geq F(s')$, $\forall s \neq s' \in S$. $F(\cdot)$ is commonly called the quality function. Furthermore, $0 < \rho \leq 1$ is the pheromone evaporation rate. This pheromone update rule aims at an increase of pheromone on solution components that have been found in high-quality solutions.

In the following, we describe the more general ACO metaheuristic, which is based on the same basic principles as AS. The ACO metaheuristic framework that is shown in Figure 17 covers all the improvements and extensions of AS which have been developed over the years. It consists of three parts gathered in the ScheduleActivities construct. The ScheduleActivities construct does not specify how these three activities are scheduled and synchronized. This is up to the algorithm designer.

AntBasedSolutionConstruction(): An ant constructively builds a solution to the problem by moving through nodes of the construction graph $\mathcal{G}$. Ants move by applying a stochastic local decision policy that makes use of the pheromone values and the heuristic values on components and/or connections of the construction graph (e.g., see the state transition rule of AS). While moving, the ant keeps in memory the partial solution it has built in terms of the path it was walking on the construction graph.

PheromoneUpdate(): When adding a component c_j to the current partial solution, an ant can update the pheromone trail(s) τ_i and/or τ_{ij} (in case the ant was walking on connection l_{ij} in order to reach component c_j). This kind of pheromone update is called *online step-by-step pheromone update*. Once an ant has built a solution, it can retrace the same path backward (by using its memory) and update

the pheromone trails of the used components and/or connections according to the quality of the solution it has built. This is called *online delayed pheromone update*. Pheromone evaporation is the process by means of which the pheromone trail intensity on the components decreases over time. From a practical point of view, pheromone evaporation is needed to avoid a too rapid convergence of the algorithm toward a sub-optimal region. It implements a useful form of *forgetting*, favoring the exploration of new areas in the search space.

DaemonActions(): Daemon actions can be used to implement centralized actions which cannot be performed by single ants. Examples are the use of a local search procedure applied to the solutions built by the ants, or the collection of global information that can be used to decide whether it is useful or not to deposit additional pheromone to bias the search process from a nonlocal perspective. As a practical example, the daemon can observe the path found by each ant in the colony and choose to deposit extra pheromone on the components used by the ant that built the best solution. Pheromone updates performed by the daemon are called *offline pheromone updates*.

Within the ACO metaheuristic framework, as shortly described above, the currently best performing versions in practise are Ant Colony System (ACS) [Dorigo and Gambardella 1997] and *MAX-MIN* Ant System (*MMAS*) [Stützle and Hoos 2000]. In the following, we are going to briefly outline the peculiarities of these algorithms.

Ant Colony System (*ACS*). The ACS algorithm has been introduced to improve the performance of AS. ACS is based on AS but presents some important differences. First, the daemon updates pheromone trails offline: At the end of an iteration of the algorithm—once all the ants have built a solution—pheromone is added to the arcs used by the ant that found the best solution from the start of the algorithm. Second, ants use a different decision rule to decide to which component to move next in the construction graph. The rule is called *pseudo-random-proportional* rule. With this rule, some moves are chosen deterministically (in a greedy manner), others are chosen probabilistically with the usual decision rule. Third, in ACS, ants perform only online step-by-step pheromone updates. These updates are performed to favor the emergence of other solutions than the best so far.

MAX-MIN Ant System (*MMAS*). *MMAS* is also an extension of AS. First, the pheromone trails are only updated offline by the daemon (the arcs that were used by the iteration best ant or the best ant since the start of the algorithm receive additional pheromone). Second, the pheromone values are restricted to an interval $[\tau_{min}, \tau_{max}]$ and the pheromone trails are initialized to their maximum value τ_{max}. Explicit bounds on the pheromone trails prevent that the probability to construct a solution falls below a certain value greater than 0. This means that the chance of finding a global optimum never vanishes during the course of the algorithm.

Recently, researchers have been dealing with finding similarities between ACO algorithms and probabilistic learning algorithms such as EDAs. An important step into this direction was the development of the Hyper-Cube Framework for Ant Colony Optimization (HC-ACO) [Blum et al. 2001]. An extensive study on this subject has been presented in Zlochin et al. [2004], where the authors present a unifying framework for so-called Model-Based Search (MBS) algorithms. Also, the close relation of algorithms like Population-Based Incremental Learning (PBIL) [Baluja and Caruana 1995] and the Univariate Marginal Distribution Algorithm (UMDA) [Mühlenbein and Paaß 1996] to ACO algorithms in the Hyper-Cube Framework has been shown. We refer the interested reader to Zlochin et al. [2004] for more information on this subject. Furthermore, connections of ACO algorithms to Stochastic Gradient Descent (SGD) algorithms are shown in Meuleau and Dorigo [2002].

Successful applications of ACO include the application to routing in communication networks [Di Caro and Dorigo

1998]; the application to the Sequential Ordering Problem (SOP) [Gambardella and Dorigo 2000], and the application to Resource Constraint Project Scheduling (RCPS) [Merkle et al. 2002]. Further references to applications of ACO can be found in Dorigo and Stützle [2002, 2003].

5. A UNIFYING VIEW ON INTENSIFICATION AND DIVERSIFICATION

In this section, we take a closer look at the concepts of *intensification* and *diversification* as the two powerful forces driving metaheuristic applications to high performance. We give a view on metaheuristics that is characterized by the way intensification and diversification are implemented. Although the relevance of these two concepts is commonly agreed, so far there is no unifying description to be found in the literature. Descriptions are very generic and metaheuristic specific. Therefore most of them can be considered incomplete and sometimes they are even opposing. Depending on the paradigm behind a particular metaheuristic, intensification and diversification are achieved in different ways. Even so, we propose a unifying view on intensification and diversification. Furthermore, this discussion could lead to the goal-directed development of hybrid algorithms combining concepts originating from different metaheuristics.

5.1. Intensification and Diversification

Every metaheuristic approach should be designed with the aim of effectively and efficiently exploring a search space. The search performed by a metaheuristic approach should be "clever" enough to both intensively explore areas of the search space with high quality solutions, and to move to unexplored areas of the search space when necessary. The concepts for reaching these goals are nowadays called intensification and diversification. These terms stem from the TS field [Glover and Laguna 1997]. In other fields—such as the EC field—related concepts are often denoted by exploitation (related to intensification) and exploration (related to diversification). However, the terms exploitation and exploration have a somewhat more restricted meaning. In fact, the notions of exploitation and exploration often refer to rather short term strategies tied to randomness, whereas intensification and diversification refer to rather medium and long term strategies based on the usage of memory. As the various different ways of using memory become increasingly important in the whole field of metaheuristics, the terms intensification and diversification are more and more adopted and understood in their original meaning.

An implicit reference to the concept of "locality" is often introduced when intensification and diversification are involved. The notion of "area" (or "region") of the search space and of "locality" can only be expressed in a fuzzy way, as they always depend on the characteristics of the search space as well as on the definition of metrics on the search space (distances between solutions).

The literature provides several high level descriptions of intensification and diversification. In the following, we cite some of them.

"Two highly important components of Tabu Search are intensification and diversification strategies. Intensification strategies are based on modifying choice rules to encourage move combinations and solution features historically found good. They may also initiate a return to attractive regions to search them more thoroughly. Since elite solutions must be recorded in order to examine their immediate neighborhoods, explicit memory is closely related to the implementation of intensification strategies. **The main difference between intensification and diversification is that during an intensification stage the search focuses on examining neighbors of elite solutions. [···] The diversification stage on the other hand encourages the search process to examine unvisited regions and to generate solutions that differ in various significant ways from those seen before.**" [Glover and Laguna 1997]

Inverted Files for Text Search Engines

JUSTIN ZOBEL

RMIT University, Australia

AND

ALISTAIR MOFFAT

The University of Melbourne, Australia

The technology underlying text search engines has advanced dramatically in the past decade. The development of a family of new index representations has led to a wide range of innovations in index storage, index construction, and query evaluation. While some of these developments have been consolidated in textbooks, many specific techniques are not widely known or the textbook descriptions are out of date. In this tutorial, we introduce the key techniques in the area, describing both a core implementation and how the core can be enhanced through a range of extensions. We conclude with a comprehensive bibliography of text indexing literature.

Categories and Subject Descriptors: H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Query formulation, retrieval models, search process*; D.4.3 [**Operating Systems**]: File Systems Management—*Access methods, file organization*; E.4 [**Coding and Information Theory**]—*Data compaction and compression*; E.5 [**Files**]—*Organization/structure*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*; I.7.3 [**index Generation**]

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Inverted file indexing, document database, text retrieval, information retrieval, Web search engine

1. INTRODUCTION

Text search is a key technology. Search engines that index the Web provide a breadth and ease of access to information that was inconceivable only a decade ago. Text search has also grown in importance at the other end of the size spectrum. For example, the help services built into operating systems rely on efficient text search, and desktop search systems help users locate files on their personal computers.

The Australian Research Council and the Victorian Partnership for Advanced Computing have provided valuable financial support as have our two home institutions.

Authors' addresses: J. Zobel, School of Computer Science & Information Technology, RMIT University, Australia; email: jz@cs.rmit.edu.au; A. Moffat, Department of Computer Science & Software Engineering, The University of Melbourne, Australia.

Search engines are structurally similar to database systems. Documents are stored in a repository, and an index is maintained. Queries are evaluated by processing the index to identify matches which are then returned to the user. However, there are also many differences. Database systems must contend with arbitrarily complex queries, whereas the vast majority of queries to search engines are lists of terms and phrases. In a database system, a match is a record that meets a specified logical condition; in a search engine, a match is a document that is appropriate to the query according to statistical heuristics and may not even contain all of the query terms. Database systems return all matching records; search engines return a fixed number of matches, which are *ranked* by their statistical similarity. Database systems assign a unique access key to each record and allow searching on that key; for querying on a Web collection, there may be many millions of documents with nonzero similarity to a query. Thus, while search engines do not have the costs associated with operations such as relational join, there are significant obstacles to fast response, that is, a query term may occur in a large number of the documents, and each document typically contains a large number of terms.

The challenges presented by text search have led to the development of a wide range of algorithms and data structures. These include representations for text indexes, index construction techniques, and algorithms for evaluation of text queries. Indexes based on these techniques are crucial to the rapid response provided by the major Web search engines. Through the use of compression and careful organization, the space needed for indexes and the time and disk traffic required during query evaluation are reduced to a small fraction of previous requirements.

In this tutorial, we explain how to implement high-performance text indexing. Rather than explore all the alternative approaches that have been described (some speculative, some ineffective, and some proven in practice), we describe a simple, effective solution that has been shown to work well in a range of contexts. This *indexing core* includes algorithms for construction of a document-level index and for basic ranked query evaluation.

Given the indexing core, we then sketch some of the principal refinements that have been devised. These include index reorganization, phrase querying, distribution, index maintenance, and, in particular, index compression. The most compelling application of the indexing techniques described in this tutorial is their use in Web search engines and so we briefly review search in the context of the Web. In the context of the different approaches, we identify significant papers and indicate their contribution.

Other research topics in text search include innovations such as metasearch, compression techniques for stored documents, and improvements to fundamental technologies such as sorting, storing, and searching of sets of strings. Domains of applicability beyond text include genomic string searching, pattern matching, and proprietary document management systems. These developments are of wide practical importance but are beyond the scope of this tutorial.

This article has a slightly unusual structure. The citations are collected into Section 13 which is a critical survey of work in the area. The earlier sections are free of citations, and reflect our desire to present the material as a tutorial that might be useful for people new to the area. To allow the corresponding citations to be accessed, the structure within Section 13 of the previous sections. That is, Section 13 can be used as a "further reading" overview for each topic.

2. TEXT SEARCH AND INFORMATION RETRIEVAL

Search engines are tools for finding the documents in a collection that are good matches to user queries. Typical kinds of document collection include Web pages, newspaper articles, academic publications, company reports, research grant applications, manual

pages, encyclopedias, parliamentary proceedings, bibliographies, historical records, electronic mail, and court transcripts.

These collections range dramatically in size. The plain text of a complete set of papers written by a researcher over ten years might occupy 10 megabytes, and the same researcher's (plain text, non-spam) 10-year email archive might occupy 100 megabytes. A thousand times bigger, the text of all the books held in a small university library might occupy around 100 gigabytes. In 2005, the complete text of the Web was probably some several tens of terabytes.

Collections also vary in the way they change over time. A newswire archive or digital library might grow only slowly, perhaps by a few thousand documents a day; deletions are rare. Web collections, in contrast, can be highly dynamic. Fortunately, many of the same search and storage techniques are useful for these collections.

Text is not the only kind of content that is stored in document collections. Research papers and newspaper articles include images, email includes attachments, and Web collections include audio and video formats. The sizes discussed previously are for text only; the indexing of media other than text is beyond the scope of this tutorial.

Query Modes. In traditional databases, the primary method of searching is by key or record identifier. Such searching is rare in text databases. Text in some kinds of collections does have structured attributes such as <author> tags and metadata such as the subject labels used for categorizing books in libraries, but these are only occasionally useful for content-based search and are not as useful as are keys in a relational database.

The dominant mode of text search is by its *content* in order to satisfy an *information need*. People search in a wide variety of ways. Perhaps the commonest mode of searching is to issue an initial query, scan a list of suggested answers, and follow pointers to specific documents. If this approach does not lead to discovery of useful documents, the user refines or modifies the query and may use advanced querying features such as restricting the search domain or forcing inclusion or omission of specific query terms. In this model of searching, an information need is represented by a *query*, and the user may issue several queries in pursuit of one information need. Users expect to be able to match documents according to any of the terms they contain.

Both casual users and professionals make extensive use of search engines but their typical strategies differ. Casual users generally examine only the first page or so returned by their favorite search engine, while professionals may use a range of search strategies and tools and are often prepared to scrutinize hundreds of potential answers. However, the same kinds of retrieval technique works for both types of searcher.

Another contrast with traditional databases is the notion of *matching*. A record matches an SQL query if the record satisfies a logical condition. A document matches an information need if the user perceives it to be *relevant*. But relevance is inexact, and a document may be relevant to an information need even though it contains none of the query terms or irrelevant even though it contains them all. Users are aware that only some of the matches returned by the system will be relevant and that different systems can return different matches for the same query. This inexactitude introduces the notion of *effectiveness*: informally, a system is effective if a good proportion of the first r matches returned are relevant. Also, different search mechanisms have different computational requirements and so measurement of system performance must thus consider both effectiveness and efficiency.

There are many ways in which effectiveness can be quantified. Two commonly used measures are *precision* and *recall*, respectively the fraction of the retrieved documents that are relevant and the fraction of relevant documents that are retrieved. There are many other metrics in use with differing virtues and differing areas of application. In

```
1   The old night keeper keeps the keep in the town
2   In the big old house in the big old gown.
3   The house in the town had the big old keep
4   Where the old night keeper never did sleep.
5   The night keeper keeps the keep in the night
6   And keeps in the dark and sleeps in the light.
```

Fig. 1. The Keeper database. It consists of six one-line documents.

this tutorial, our focus is on describing indexing techniques that are efficient, and we do not review the topic of effectiveness. Worth noting, however, is that the research that led to these efficient indexing techniques included demonstrations that they do not compromise effectiveness.

With typical search engines, the great majority of information needs are presented as *bag-of-word* queries. Many bag-of-word queries are in fact *phrases* such as proper names. Some queries have phrases marked up explicitly, in quotes. Another common approach is to use Boolean operators such as AND, perhaps to restrict answers to a specific language, or to require that all query terms must be present in an answer.

Example Collections. A sample collection, used as an example through this tutorial, is shown in Figure 1. In this Keeper database, only document 2 is about a big old house. But with a simple matching algorithm, the bag-of-words query `big old house` matches documents 2 and 3, and perhaps also documents 1 and 4 which contain `old`, but not the other terms. The phrase query `"big old house"` would match only document 2.

Another issue is the parsing method used to extract terms from text. For example, when HTML documents are being indexed, should the markup tags be indexed? or terms within tags? And should hyphenated terms be considered as one word or two? An even more elementary issue is whether to *stem* and *casefold*, that is, remove variant endings from words, and convert to lowercase. Choice of parsing technique has little impact, however, on indexing. On the other hand, the issue of *stopping* does affect the cost of indexing and removal of common words or function words such as the and `furthermore` can have a significant effect. Confusingly, in some information retrieval literature, the task of parsing is known as index term extraction or simply indexing. In this article, indexing is the task of constructing an index.

Without stemming, but with casefolding, the vocabulary of Keeper is:

```
and big dark did gown had house in keep keeper keeps light
never night old sleep sleeps the town where
```

Stemming might reduce the vocabulary to:

```
and big dark did gown had house in keep light never night old
sleep the town where
```

with the exact result dependent on the stemming method used. Stopping might then reduce the Keeper vocabulary to:

```
big dark gown house keep light night old sleep town
```

In addition to the example Keeper collection, two hypothetical collections are used to illustrate efficiency issues. The characteristics of these collections are shown in Table I, and, while they are not actual data sets, they are based on experience with real text. Specifically, they are similar to two types of collections provided by the TREC project run by the United States National Institute for Standards and Technology (NIST) (see `trec.nist.gov`). TREC has been a catalyst for research in information retrieval since 1992, and without it, robust measurement of the techniques described in this tutorial

Table I. Characteristics of Two Hypothetical Text Databases, Used as Examples in This Tutorial

	NewsWire	Web
Size (gigabytes)	1	100
Documents	400,000	12,000,000
Word occurrences (without markup)	180,000,000	11,000,000,000
Distinct words (after stemming)...,	400,000	16,000,000
per document, totaled	70,000,000	3,500,000,000

would have been difficult or impossible. The last line in the table is the number of distinct word-document pairs, that is, the number of word occurrences when duplicates within a document are not counted.

Most of the costs required for additional structures scale more or less linearly for collections larger than a gigabyte. For example, in Web data, new distinct words continue to occur at a typical rates of about one per 500–1000 word occurrences. Typical query terms occur in 0.1%–1% of the indexed documents.

Taking all of these factors into account, implementors of search engines must design their systems to balance a range of technical requirements:

—effective resolution of queries;

—use of features of conventional text, such as query term proximity, that improve effectiveness;

—use of features of hyperlinked text, such as anchor strings and URL terms, that improve effectiveness;

—fast resolution of queries;

—minimal use of other resources (disk, memory, bandwidth);

—scaling to large volumes of data;

—change in the set of documents; and

—provision of advanced features such as Boolean restriction and phrase querying.

This tutorial describes techniques for supporting all of these requirements.

Similarity Measures. All current search engines use *ranking* to identify potential answers. In a ranked query, a statistical similarity heuristic or *similarity measure* is used to assess the closeness of each document to the textual query. The underlying principle is that the higher the similarity score awarded to a document, the greater the estimated likelihood that a human would judge it to be relevant. The r "most similar according to the heuristic" documents are returned to the user as suggested answers. To describe the implementation of text retrieval, we first consider evaluation of bag-of-words queries in which similarity is determined by simple statistics. In Section 4, we extend these techniques to phrase queries.

Most similarity measures use some composition of a small number of fundamental statistical values:

— $f_{d,t}$, the frequency of term t in document d;

— $f_{q,t}$, the frequency of term t in the query;

— f_t, the number of documents containing one or more occurrences of term t;

— F_t, the number of occurrences of term t in the collection;

— N, the number of documents in the collection; and

— n, the number of indexed terms in the collection.

These basic values are combined in a way that results in three monotonicity observations being enforced.

(1) Less weight is given to terms that appear in many documents;
(2) More weight is given to terms that appear many times in a document; and
(3) Less weight is given to documents that contain many terms.

The intention is to bias the score towards relevant documents by favoring terms that appear to be discriminatory and reducing the impact of terms that appear to be randomly distributed.

A typical older formulation that is effective in practice calculates the cosine of the angle in n-dimensional space between a query vector $\langle w_{q,t} \rangle$ and a document vector $\langle w_{d,t} \rangle$. There are many variations of the cosine formulation. An example is:

$$w_{q,t} = \ln\left(1 + \frac{N}{f_t}\right) \qquad w_{d,t} = 1 + \ln f_{d,t}$$
$$W_d = \sqrt{\sum_t w_{d,t}^2} \qquad W_q = \sqrt{\sum_t w_{q,t}^2} \tag{1}$$
$$S_{q,d} = \frac{\sum_t w_{d,t} \cdot w_{q,t}}{W_d \cdot W_q}.$$

The term W_q can be neglected as it is a constant for a given query and does not affect the ordering of documents. Variations on this theme are to avoid use of logarithms, or replace N by $\max_t\{f_t\}$ in the expression for $w_{q,t}$, or multiply the query-term weights $w_{q,t}$ by $1 + \ln f_{q,t}$ when queries are long, or use a different way of combining $w_{q,t}$ and $w_{d,t}$, or take W_d to be the length of the document in words or in bytes, and so on.

What all of these variants share is that the quantity $w_{q,t}$ typically captures the property often described as the inverse document frequency of the term, or IDF, while $w_{d,t}$ captures the term frequency, or TF, hence the common description of similarity measures as TF×IDF formulations. Observing that the negative log of a probability is the information content, the score assigned to a document can very loosely be interpreted from an entropy-based perspective as being a sum of information conveyed by the query terms in that document, which maps to the same ordering as the product of their respective probabilities.

Similarity formulations that are directly grounded in statistical principles have also proven successful in TREC. The best known of these is the Okapi computation,

$$w_{q,t} = \ln\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \cdot \frac{(k_3 + 1) \cdot f_{q,t}}{k_3 + f_{q,t}}$$
$$w_{d,t} = \frac{(k_1 + 1) f_{d,t}}{K_d + f_{d,t}}$$
$$K_d = k_1\left((1 - b) + b\frac{W_d}{W_A}\right) \tag{2}$$
$$S_{q,d} = \sum_{t \in q} w_{q,t} \cdot w_{d,t},$$

in which the values k_1 and b are parameters, set to 1.2 and 0.75 respectively; k_3 is a parameter that is set to ∞, so that the expression $(k_3 + 1) \cdot f_{q,t} / (k_3 + f_{q,t})$ is assumed to be equivalent to $f_{q,t}$; and W_d and W_A are the document length and average document length, in any suitable units.

To rank a document collection with regard to a query q and identify the top r matching documents:

(1) Calculate $w_{q,t}$ for each query term t in q.
(2) For each document d in the collection,
  (a) Set $S_d \leftarrow 0$.
  (b) For each query term t,
    Calculate or read $w_{d,t}$, and
    Set $S_d \leftarrow S_d + w_{q,t} \times w_{d,t}$.
  (c) Calculate or read W_d.
  (d) Set $S_d \leftarrow S_d / W_d$.
(3) Identify the r greatest S_d values and return the corresponding documents.

Fig. 2. Exhaustive computation of cosine similarity between a query q and every document in a text collection. This approach is suitable only when the collection is small or is highly dynamic relative to the query rate.

More recent probabilistic approaches are based on language models. There are many variants; a straightforward language-model formulation is:

$$w_{d,t} = \log\left(\frac{|d|}{|d|+\mu} \cdot \frac{f_{d,t}}{|d|} + \frac{\mu}{|d|+\mu} \cdot \frac{F_t}{|C|}\right) \tag{3}$$

$$S_{q,d} = \sum_{t \in q} f_{q,t} \cdot w_{d,t},$$

where $|d|$ (respectively, $|C|$) is the number of term occurrences in document d (respectively, collection C) and μ is a parameter, typically set to 2500. The left-hand side of the sum is the observed likelihood of the term in the document, while the right-hand side modifies this (using Dirichlet smoothing) by combining it with the observed likelihood of the term in the collection. Taking the smoothed value as an estimate of the probability of the term in the document, this formulation is rank equivalent to ordering documents by the extent to which the query's entropy in the document's model differs from the query's entropy in the collection as a whole.

In this language-model formulation, the value $w_{d,t}$ is nonzero even if t is not in d, presenting difficulties for the term-ordered query evaluation strategies we explain later. However, this problem can be addressed by transforming it into a rank-equivalent measure:

$$S_{q,d} \stackrel{rank}{=} |q| \cdot \log\left(\frac{\mu}{|d|+\mu}\right) + \sum_{t \in q \wedge d} \left(f_{q,t} \cdot \log\left(\frac{f_{d,t}}{\mu} \cdot \frac{|C|}{F_t} + 1\right)\right), \tag{4}$$

where the term-oriented component $\log(1 + (f_{d,t}/\mu) \cdot (|C|/F_t))$ is zero when t is not in d.

In all of these formulations, documents can score highly even if some of the query terms are missing. This is a common attribute of similarity heuristics. We do not explore similarity formulations in detail and take as our brief the need to implement a system in which any such computation can be efficiently computed.

Given a formulation, ranking a query against a collection of documents is in principle straightforward: each document is fetched in turn, and the similarity between it and the query calculated. The documents with the highest similarities can then be returned to the user. An algorithm for exhaustive ranking using the cosine measure is shown in Figure 2.

The drawback of this approach is that every document is explicitly considered, but for typical situations in which $r \ll N$, only a tiny fraction of documents are returned as answers. For most documents, the vast majority of similarity values are

insignificant. The exhaustive approach does, however, illustrate the main features of computing a ranking, and, with a simple reorganization, a more efficient computation can be achieved based on the key observation that, to have a nonzero score, a document must contain at least one query term.

3. INDEXING AND QUERY EVALUATION

Fast query evaluation makes use of an *index*: a data structure that maps terms to the documents that contain them. For example, the index of a book maps a set of selected terms to page numbers. With an index, query processing can be restricted to documents that contain at least one of the query terms.

Many different types of index have been described. The most efficient index structure for text query evaluation is the *inverted file*: a collection of lists, one per term, recording the identifiers of the documents containing that term. Other structures are briefly considered in Section 11, but they are not useful for general-purpose querying.

Baseline Inverted File. An inverted file index consists of two major components. The *search structure* or *vocabulary* stores for each distinct word t,

—a count f_t of the documents containing t, and
—a pointer to the start of the corresponding *inverted list*.

Studies of retrieval effectiveness show that all terms should be indexed, even numbers. In particular, experience with Web collections shows that any visible component of a page might reasonably be used as a query term, including elements such as the tokens in the URL. Even stopwords—which are of questionable value for bag-of-words queries—have an important role in phrase queries.

The second component of the index is a set of *inverted lists* where each list stores for the corresponding word t,

—the identifiers d of documents containing t, represented as ordinal document numbers; and
—the associated set of frequencies $f_{d,t}$ of terms t in document d.

The lists are represented as sequences of $(d, f_{d,t})$ pairs. As described, this is a *document-level* index in that word positions within documents are not recorded. Together with an array of W_d values (stored separately), these components provide all the information required for both Boolean and ranked query evaluation. A complete inverted file for the `Keeper` database is shown in Figure 3.

In a complete text database system, there are several other structures, including the documents themselves and a table that maps ordinal document numbers to disk locations (or other forms of document locator such as a filename or other key). We do not explore these structures in this tutorial.

In a simple representation, for the `NewsWire` data, the total index size would be approximately 435MB (megabytes) or around 40% of the size of the original data. This is comprised of 12MB for 400,000 words, pointers, and counts; 1.6MB for 400,000 W_d values; 280MB for 70,000,000 document identifiers (four bytes each); and 140MB for 70,000,000 document frequencies (two bytes each). For the `Web` data, the total size is about 21GB (gigabytes) or just over 20% of the original text. The difference between these two collections is a consequence of Web pages tending to contain large volumes of unindexed markup.

An assumption made in these calculations is that each inverted list is stored contiguously. The alternative is that lists are composed of a sequence of blocks that are linked or indexed in some way. The assumption of contiguity has a range of implications. First,

term t	f_t	Inverted list for t
and	1	⟨6, 2⟩
big	2	⟨2, 2⟩ ⟨3, 1⟩
dark	1	⟨6, 1⟩
did	1	⟨4, 1⟩
gown	1	⟨2, 1⟩
had	1	⟨3, 1⟩
house	2	⟨2, 1⟩ ⟨3, 1⟩
in	5	⟨1, 1⟩ ⟨2, 2⟩ ⟨3, 1⟩ ⟨5, 1⟩ ⟨6, 2⟩
keep	3	⟨1, 1⟩ ⟨3, 1⟩ ⟨5, 1⟩
keeper	3	⟨1, 1⟩ ⟨4, 1⟩ ⟨5, 1⟩
keeps	3	⟨1, 1⟩ ⟨5, 1⟩ ⟨6, 1⟩
light	1	⟨6, 1⟩
never	1	⟨4, 1⟩
night	3	⟨1, 1⟩ ⟨4, 1⟩ ⟨5, 2⟩
old	4	⟨1, 1⟩ ⟨2, 2⟩ ⟨3, 1⟩ ⟨4, 1⟩
sleep	1	⟨4, 1⟩
sleeps	1	⟨6, 1⟩
the	6	⟨1, 3⟩ ⟨2, 2⟩ ⟨3, 3⟩ ⟨4, 1⟩ ⟨5, 3⟩ ⟨6, 2⟩
town	2	⟨1, 1⟩ ⟨3, 1⟩
where	1	⟨4, 1⟩

d	1	2	3	4	5	6
W_d	11.4	13.5	11.4	8.0	11.3	12.6

Fig. 3. Complete document-level inverted file for the Keeper database. The entry for each term t is composed of the frequency f_t and a list of pairs, each consisting of a document identifier d and a document frequency $f_{d,t}$. Also shown are the W_d values as computed for the cosine measure shown in Equation 1.

it means that a list can be read or written in a single operation. Accessing a sequence of blocks scattered across a disk would impose significant costs on query evaluation as the list for a typical query term on the Web data would occupy 100kB (kilobytes) to 1MB, and the inverted list for a common term could be many times this size. Adding to the difficulties for the great majority of terms, the inverted list is much less than a kilobyte, placing a severe constraint on feasible size for a fixed-size block. Second, no additional space is required for next-block pointers. Third, index update procedures must manage variable-length fragments that vary enormously in size, from tiny to vast; our experience, however, is that the benefits of contiguity greatly outweigh these costs.

An issue that is considered in detail in Section 8 is how to represent each stored value such as document numbers and in-document frequencies. The choice of any fixed number of bits or bytes to represent a value is clearly arbitrary and has potential implications for scaling (fixed-length values can overflow) and efficiency (inflation in the volume of data to be managed). Using the methods described later in this article, large gains in performance are available through the use of compressed representations of indexes.

To facilitate compression, d-gaps are stored rather than straight document identifiers. For example, the sorted sequence of document numbers

$$7, 18, 19, 22, 23, 25, 63, \ldots$$

can be represented by gaps

$$7, 11, 1, 3, 1, 2, 38, \ldots .$$

To use an inverted index to rank a document collection with regard to a query q and identify the top r matching documents:

(1) Allocate an accumulator A_d for each document d and set $A_d \leftarrow 0$.
(2) For each query term t in q,
  (a) Calculate $w_{q,t}$, and fetch the inverted list for t.
  (b) For each pair $\langle d, f_{d,t} \rangle$ in the inverted list
    Calculate $w_{d,t}$, and
    Set $A_d \leftarrow A_d + w_{q,t} \times w_{d,t}$.
(3) Read the array of W_d values.
(4) For each $A_d > 0$, set $S_d \leftarrow A_d / W_d$.
(5) Identify the r greatest S_d values and return the corresponding documents.

Fig. 4. Indexed computation of cosine similarity between a query q and a text collection.

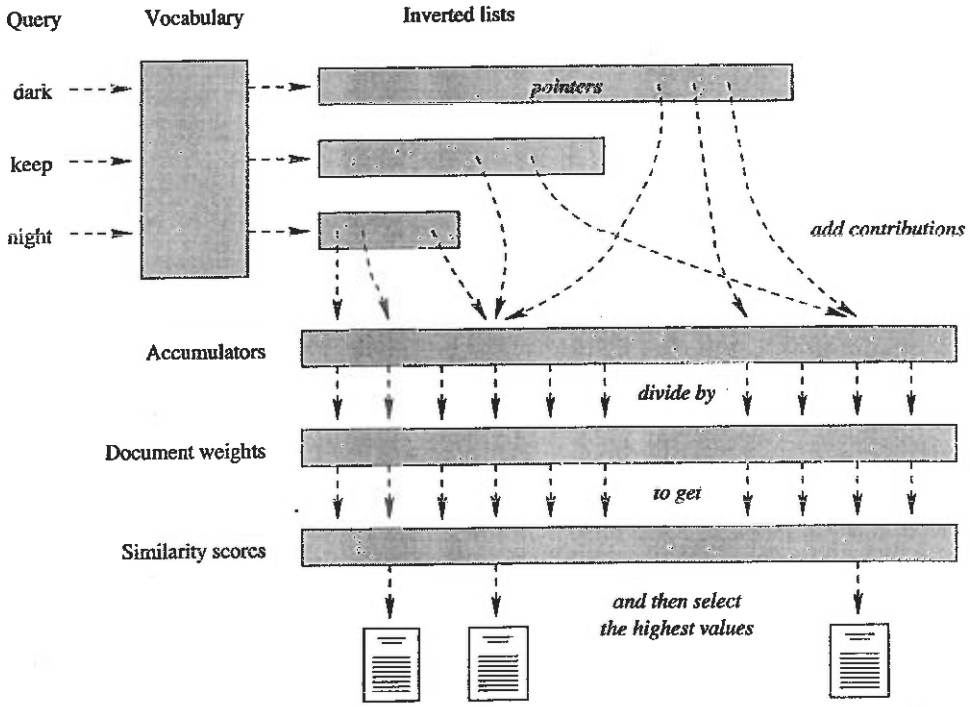

Fig. 5. Using an inverted file and a set of accumulators to calculate document similarity scores.

While this transformation does not reduce the maximum magnitude of the stored numbers, it does reduce the average, providing leverage for the compression techniques discussed later. Section 8 gives details of mechanisms that can exploit the advantage that is created by gaps.

Baseline Query Evaluation. Ranking using an inverted file is described in Figure 4 and illustrated in Figure 5. In this algorithm, the query terms are processed one at a time. Initially each document has a similarity of zero to the query; this is represented by creating an array A of N partial similarity scores referred to as *accumulators*, one for each document d. Then, for each term t, the accumulator A_d for each document d mentioned in t's inverted list is increased by the contribution of t to the similarity of d

to the query. Once all query terms have been processed, similarity scores S_d are calculated by dividing each accumulator value by the corresponding value of W_d. Finally, the r largest similarities are identified, and the corresponding documents returned to the user.

The cost of ranking via an index is far less than with the exhaustive algorithm outlined in Figure 2. Given a query of three terms, processing a query against the `Web` data involves finding the three terms in the vocabulary; fetching and then processing three inverted lists of perhaps 100kB to 1MB each; and making two linear passes over an array of 12,000,000 accumulators. The complete sequence requires well under a second on current desktop machines.

Nonetheless, the costs are still significant. *Disk space* is required for the index at 20%–60% of the size of the data for an index of the type shown in Figure 3; *memory* is required for an accumulator for each document and for some or all of the vocabulary; *CPU time* is required for processing inverted lists and accumulators; and *disk traffic* is used to fetch inverted lists. Fortunately, compared to the implementation shown in Figure 4, all of these costs can be dramatically reduced.

Indexing Word Positions. We have described inverted lists as sequences of index entries, each a $\langle d, f_{d,t} \rangle$ pair. An index of this form is *document-level* since it indicates whether a term occurs in a document but does not contain information about precisely where the term appears. Given that the frequency $f_{d,t}$ represents the number of occurrences of t in d, it is straightforward to modify each entry to include the $f_{d,t}$ ordinal word positions p at which t occurs in d and create a *word-level* inverted list containing pointers of the form $\langle d, f_{d,t}, p_1, \ldots, p_{f_{d,t}} \rangle$. Note that in this representation positions are word counts, not byte counts, so that they can be used to determine adjacency.

Word positions can be used in a variety of ways during query evaluation. Section 4 discusses one of these, phrase queries in which the user can request documents with a sequence rather than bag-of-words. Word positions can also be used in bag-of-word queries, for example, to prefer documents where the terms are close together or are close to the beginning of the document. Similarity measures that make use of such proximity mechanisms have not been particularly successful in experimental settings but, for simple queries, adjacency and proximity do appear to be of value in Web retrieval.

If the source document has a hierarchical structure, that structure can be reflected by a similar hierarchy in the inverted index. For example, a document with a structure of chapters, sections, and paragraphs might have word locations stored as (c, s, p, w) tuples coded as a sequence of nested runs of c-gaps, s-gaps, p-gaps, and w-gaps. Such an index allows within-same-paragraph queries as well as phrase queries, for example, and with an appropriate representation, is only slightly more expensive to store than a nonhierarchical index.

Core Ideas. To end this section, we state several key implementation decisions.

—Documents have ordinal identifiers, numbered from one.

—Inverted lists are stored contiguously.

—The vocabulary consists of every term occurring in the documents and is stored in a simple extensible structure such as a B-tree.

—An inverted list consists of a sequence of pairs of document numbers and in-document frequencies, potentially augmented by word positions.

—The vocabulary may be preprocessed, by stemming and stopping.

—Ranking involves a set of accumulators and term-by-term processing of inverted lists.

This set of choices constitutes a core implementation in that it provides an approach that is simple to implement and has been used in several public-domain search systems and, we believe, many proprietary systems. In the following sections, we explore extensions to the core implementation.

4. PHRASE QUERYING

A small but significant fraction of the queries presented to Web search engines include an explicit phrase, such as `"philip glass" opera` or `"the great flydini"`. Users also often enter phrases without explicit quotes, issuing queries such as `Albert Einstein` or `San Francisco hotel`. Intuitively it is appealing to give high scores to pages in which terms appear in the same order and pattern as they appear in the query, and low scores to pages in which the terms are separated.

When phrases are used in Boolean queries, it is clear what is intended—the phrase itself must exist in matching documents. For example, the Boolean query `old "night keeper"` would be evaluated as if it contains two query terms, one of which is a phrase, and both terms would be required for a document to match.

In a ranked query, a phrase can be treated as an ordinary term, that is, a lexical entity that occurs in given documents with given frequencies, and contributes to the similarity score for that document when it does appear. Similarity can therefore be computed in the usual way, but it is first necessary to use the inverted lists for the terms in the phrase to construct an inverted list for the phrase itself, using a Boolean intersection algorithm. A question for information retrieval research (and outside the scope of this tutorial) is whether this is the best way to use phrases in similarity estimation. A good question for this tutorial is how to find—in a strictly Boolean sense—the documents in which a given sequence of words occur together as a phrase since, regardless of how they are eventually incorporated into a matching or ranking scheme, identification of phrases is the first step.

An obvious possibility is to use a parser at index construction time that recognizes phrases that might be queried and to index them as if they were ordinary document terms. The set of identified phrases would be added to the vocabulary and have their own inverted lists; users would then be able to query them without any alteration to query evaluation procedures. However, such indexing is potentially expensive. There is no obvious mechanism for accurately identifying which phrases might be used in queries, and the number of candidate phrases is enormous since even the number of distinct two-word phrases grows far more rapidly than the number of distinct terms. The hypothetical `Web` collection shown in Table I could easily contain a billion distinct two-word pairs.

Three main strategies for Boolean phrase query evaluation have been developed.

—Process phrase queries as Boolean bags-of-words so that the terms can occur anywhere in matching document, then postprocess the retrieved documents to eliminate false matches.

—Add word positions to some or all of the index entries so that the locations of terms in documents can be checked during query evaluation.

—Use some form of partial phrase index or word-pair index so that phrase appearances can be directly identified.

These three strategies can complement each other. However, a pure bag-of-words approach is unlikely to be satisfactory since the cost of fetching just a few nonmatching documents could exceed all other costs combined, and, for many phrases, only a tiny fraction of the documents that contain the query words also contain them as a phrase.

Phrase Query Evaluation. When only document-level querying is required, inclusion of positional information in the index not only takes space, but also slows query processing because of the need to skip over the positional information in each pointer. And, as discussed in more detail in the following, if bag-of-words ranked queries are to be supported efficiently, then other index organizations, such as frequency- and impact-sorted arrangements, need to be considered.

Taken together, these considerations suggest that maintenance of two separate indexes may be attractive, a word-level index for Boolean searching and phrase identification, and a document-level impact- or frequency-sorted index for processing ranked queries. Given that document-level indexes are small, the space overhead of having multiple indexes is low. Separating the indexes also brings flexibility and allows consideration of index structures designed explicitly for phrases.

Many phrases include common words. The cost of phrase query processing using a word-level inverted index is then dominated by the cost of fetching and decoding lists for those words which typically occur at the start of or in the middle of a phrase—consider `"the house in the town"`, for example. One way of avoiding this problem would be to neglect certain stop words and index them at the document-level only. For example, to evaluate the query `"the house in the town"` processing could proceed by intersecting the lists for `house`, and `town`, looking for positions p of `house` such that `town` is at $p + 3$. False matches could be eliminated by post-processing, that is, by fetching candidate documents and examining them directly. The possibility of false matches could also simply be ignored. However, in some phrase queries, the common words play an important semantic role and must be included.

Phrase Indexes. It is also possible to build a complete index of two-word phrases using a hierarchical storage structure to avoid an overly large vocabulary. Experiments show that such an index occupies around 50% of the size of the source data, that is, perhaps 50GB for the `Web` collection. The inverted lists for phrases are on average much shorter than those of the individual words, but there are many more of them, and the vocabulary is also much bigger.

Using a two-word phrase index, evaluation of the phrase query `"the house in the town"` (which matches line three of the `Keeper` collection) involves processing the inverted lists for, say, the phrases `"the house"`, `"house in"`, and `"the town"`. The pair `"in the"` is also a phrase but is covered by the others and—making an arbitrary choice between this phrase and `"house in"`—its inverted list does not need to be processed.

For phrases composed of rare words, having a phrase index yields little advantage, as processing savings are offset by the need to access a much larger vocabulary. A successful strategy is to have an index for word pairs that begin with a common word and combine it with a word-level inverted index. For example, the preceding query could be evaluated by intersecting the inverted lists for `"the house"`, `in`, and `"the town"`. An index of all word pairs beginning with any one of the three commonest words is about 1% of the size of the `Web` data but allows phrase querying time to be approximately halved.

5. INDEX CONSTRUCTION

The single key problem that makes index construction challenging is that the volume of data involved cannot be held in main memory in a dynamic data structure of the kind typically used for cross-reference generation. The underlying task that needs to be performed is essentially that of matrix transposition. But the documents–terms matrix is very sparse, and is far too large to be manipulated directly as an array. Instead, index construction techniques make use of index compression methods and either distributive or comparison-based sorting techniques.

To build an inverted index using the in-memory technique:

(1) Make an initial pass over the collection.
For each term t count its document frequency f_t, and determine an upper bound u_t on the length of the inverted list for t.

(2) Allocate an in-memory array of $\sum_t u_t$ bytes, and, for each term t, create a pointer c_t to the start of a corresponding block of u_t bytes.

(3) Process the collection a second time.
For each document d, and for each term t in d, append a code representing d and $f_{d,t}$ at c_t, and update c_t.

(4) Make a sequential pass over the in-memory index that has been constructed.
For each t, copy the f_t representations of the $\langle d, f_{d,t} \rangle$ pointers from the allocated u_t bytes to the inverted file, compressing if desired.

Fig. 6. In-memory inversion.

In-Memory Inversion. A simple in-memory inversion algorithm is shown in Figure 6. The key idea is that a first pass through the documents collects term frequency information, sufficient for the inverted index to be laid out in memory in template form. A second pass then places pointers into their correct positions in the template, making use of the random-access capabilities of main memory. The advantage of this approach is that almost no memory is wasted compared to the final inverted file size since there is negligible fragmentation. In addition, if compression is used, the index can be represented compactly throughout the process. This technique is viable whenever the main memory available is about 10%–20% greater than the combined size of the index and vocabulary that are to be produced. It is straightforward to extend the in-memory algorithm to include word positions, but the correspondingly larger final index will more quickly challenge the memory capacity of whatever hardware is being used since individual list entries may become many kilobytes long.

It is also possible to extend the in-memory technique to data collections where index size exceeds memory size by laying out the index skeleton on disk, creating a sequence of partial indexes in memory, and then transferring each in a skip-sequential manner to a template that has been laid out as a disk file. With this extended method, and making use of compression, indexes can be built for multi-gigabyte collections using around 10–20MB of memory beyond the space required for a dynamic vocabulary.

Sort-Based Inversion. A shortcoming of two-pass techniques of the kind sketched in Figure 6 is that document parsing and fetching is a significant component of index construction costs, perhaps half to two-thirds of the total time for Web data. The documents could be stored parsed during index construction, but doing so implies substantial disk overheads, and the need to write the parsed text may outweigh the cost of the second parsing.

Other index construction methods are based on explicit *sorting*. In a simple form of this approach, an array or file of $\langle t, d, f_{d,t} \rangle$ triples is created in document number order, sorted into term order, and then used to generate the final inverted file.

With careful sequencing and use of a multiway merge, the sort can be carried out in-place on disk using compressed blocks. The disk space overhead is again about 10% of the final compressed index size, and memory requirements and speed are also similar to partitioned inversion. As for partitioned inversion, the complete vocabulary must be kept in memory, limiting the volume of data that can be indexed on a single machine.

Merge-Based Inversion. As the volumes of disk and data grow, the cost of keeping the complete vocabulary in memory is increasingly significant. Eventually, the index must be created as an amalgam of smaller parts, each of which is constructed using one of the

To build an inverted index using a merge-based technique:

(1) Until all documents have been processed,
  (a) Initialize an in-memory index, using a dynamic structure for the vocabularies and a static coding scheme for inverted lists; store lists either in dynamically resized arrays or in linked blocks.
  (b) Read documents and insert $\langle d, f_{d,t} \rangle$ pointers into the in-memory index, continuing until all allocated memory is consumed.
  (c) Flush this temporary index to disk, including its vocabulary.
(2) Merge the set of partial indexes to form a single index, compressing the inverted lists if required.

Fig. 7. Merge-based inversion.

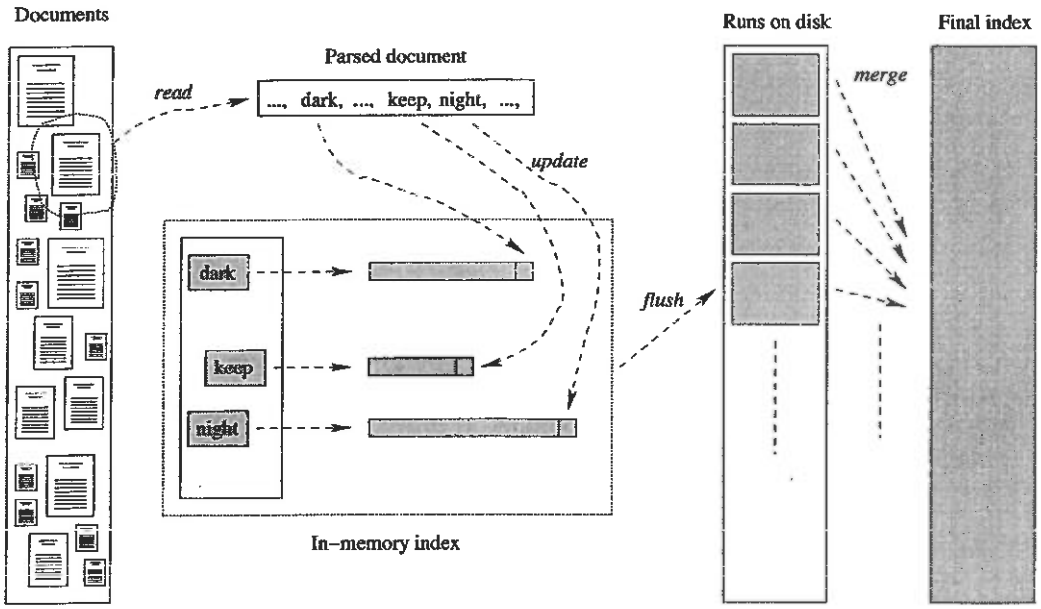

Fig. 8. Merge-build process.

previous techniques or using purely in-memory structures. Figures 7 and 8 illustrate this process.

In merge-based inversion, documents are read and indexed in memory until a fixed capacity is reached. Each inverted list needs to be represented in a structure that can grow as further information about the term is encountered, and dynamically resizable arrays are the best choice. When memory is full, the index (including its vocabulary) is flushed to disk as a single *run* with the inverted lists in the run stored in lexicographic order to facilitate subsequent merging. As runs are never queried, the vocabulary of a run does not need to be stored as an explicit structure; each term can, for example, be written at the head of its inverted list. Once the run is written, it is entirely deleted from memory so that construction of the next run begins with an initially empty vocabulary.

When all documents have been processed, the runs are merged to give the final index. The merging process builds the final vocabulary on the fly and, if a large read buffer is allocated to each run, is highly efficient in terms of disk accesses. If disk space is scarce, the final index can be written back into the space occupied by the runs as they are processed as the final index is typically a little smaller than the runs—vocabulary

information is not duplicated, and the final inverted lists can be represented more efficiently.

Merge-based index construction is practical for collections of all sizes. In particular, it scales well and operates effectively in as little as 100MB of memory. In addition, disk space overheads can be restricted to a small fraction of the final index; only one parsing pass is required over the data; and the method extends naturally to phrase indexing. Finally, the compression techniques described in Section 8 can further reduce the cost of index construction by reducing the number of runs required.

6. INDEX MAINTENANCE

Inserting one document into a text databases involves, in principle, adding a few bytes to the end of every inverted list corresponding to a term in the document. For a document of reasonable size, such an insertion involves fetching and slightly extending several hundred inverted lists and is likely to require 10–20 seconds on current hardware. In contrast, with merge-based inversion, the same hardware can index around 1,000 documents per second. That is, there is a 10,000-fold disparity in cost between these two approaches.

For fast insertion, it is necessary to avoid accessing the disk-resident inverted lists of each term. The only practical solution is to amortize the update costs over a sequence of insertions. The properties of text databases, fortunately, allow several strategies for cost amortization. In particular, for ranking, it is not always necessary for new documents to be immediately available for searches. If they are to be searchable, new documents can be made available through a temporary in-memory index—in effect, the last subindex in the merging strategy.

Three broad categories of update strategies are available: rebuild from scratch, merge an existing index with an index of new documents, and incremental update.

Rebuild. In some applications, the index may not to be updated online at all. Instead, it can be periodically rebuilt from scratch. Consider, for example, a university Web site. New documents are only discovered through crawling and immediate update is not essential. For a gigabyte of data, rebuilding takes just a few minutes, a small cost compared to that of fetching the documents to index.

Intermittent Merge. The inverted lists for even large numbers of documents can easily be maintained in the memory of a standard desktop computer. If the lists are in memory, it is cheap to insert new documents as they arrive; indeed, there is no difference between maintaining such lists and the algorithm described in Figure 7.

If existing documents are indexed in a standard inverted file and new documents are indexed in memory, the two indexes can share a common vocabulary, and all documents can be made available to queries via a straightforward adaptation of the methods described earlier. Then when memory is full, or some other criterion is met, the in-memory index is merged with the on-disk index. The old index can be used until the merge is complete, at the cost of maintaining two complete copies of the inverted lists. During the merge either a new in-memory index must be created or insertions must temporarily be blocked, and thus, during the merge, new documents are only available via exhaustive search.

It was argued earlier that inverted lists should be stored contiguously as accessing a large number of blocks would be a dominant cost of query evaluation. However, if the number of blocks is constrained (for instance, an average of three per term), the time to evaluate queries can similarly be controlled. In addition, if the index is arranged as a sequence of subindexes, each one no greater than a given fraction of the size of the next (that is, the sizes form a geometric sequence), then only a small part of the index is involved in most merge operations. This combination of techniques allow an index to

be built incrementally, and be simultaneously available for querying, in just twice the time required by an offline merge-based build.

Incremental Update. A final alternative is to update the main index term by term, as and when opportunity arises, with some terms' in-memory lists covering more documents than others'. This process is similar to the mechanisms used for maintaining variable-length records in a conventional database system. In such an asynchronous merge, a list is fetched from disk, the new information is integrated into the list, and the list is then written back. Using standard free-space management, the list can either be written back in place or, if there is insufficient space, written to a new location.

The per-list updates should be deferred for as long as possible to minimize the number of times each list is accessed. The simplest approach is to process as for the merging strategy and, when a memory limit is reached, then proceed through the whole index, amending each list in turn. Other possibilities are to update a list only when it is fetched in response to a query or to employ a background process that slowly cycles through the in-memory index, continuously updating entries. In practice, these methods are not as efficient as intermittent merge, which processes data on disk sequentially.

Choosing an Update Strategy. For reasonable collection sizes, merging is the most efficient strategy for update but has the drawback of requiring significant disk overheads. It allows relatively simple recovery as reconstruction requires only a copy of the index and the new documents. In contrast, incremental update proceeds in place with some space lost due to fragmentation. But recovery in an incremental index may be complex due to the need to track which inverted lists have been modified.

For smaller, relatively static collections, the cost of rebuild may exceed that of other methods, but still be of little consequence compared to the other costs of maintaining the database. And if the collection is highly dynamic, such as a Web site in which documents are edited as well as inserted, then inserting or deleting a single word in a document may affect all the word positions, for example, and rebuild may be the only plausible option.

7. DISTRIBUTED INFORMATION RETRIEVAL

When large volumes of data are involved or when high query volumes must be supported, one machine may be inadequate to support the load even when the various enhancements surveyed earlier are incorporated. For example, in mid-2004, the Google search engine processed more than 200 million queries a day against more than 20TB of crawled data, using more than 20,000 computers.

To handle the load, a combination of distribution and replication is required. *Distribution* refers to the fact that the document collection and its index are split across multiple machines and that answers to the query as a whole must be synthesized from the various collection components. *Replication* (or *mirroring*) then involves making enough identical copies of the system so that the required query load can be handled.

Document-Distributed Architectures. The simplest distribution regime is to partition the collection and allocate one subcollection to each of the processors. A local index is built for each subcollection; when queries arrive, they are passed to every subcollection and evaluated against every local index. The sets of subcollection answers are then combined in some way to provide an overall set of answers. The advantages of such a *document partitioned* system are several; collection growth is accommodated by designating one of the hosts as the dynamic collection so that only it needs to rebuild its index; and the computationally expensive parts of the process are distributed equally across all of the hosts in the computer cluster. The dashed region

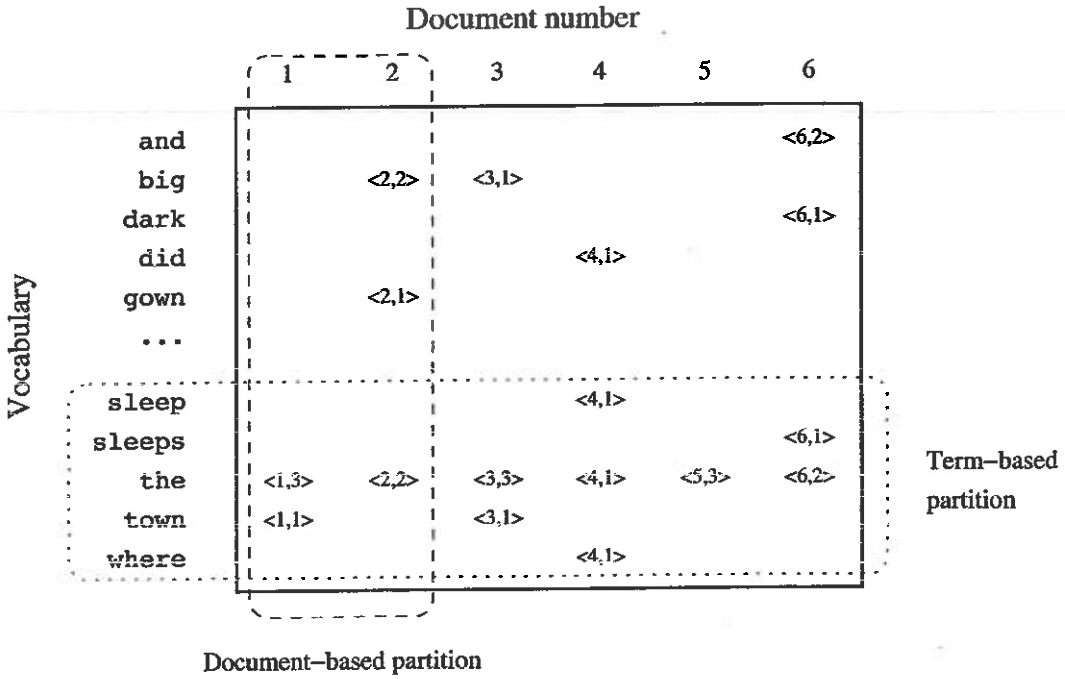

Fig. 9. Two ways in which the index of Figure 3 might be partitioned and distributed across a cluster of machines.

in Figure 9 shows one component of a document-distributed retrieval system with this one processor indexing all terms that appear in the first two documents of the collection.

Term-Distributed Architectures. An alternative strategy is *term partitioning*. In a term-partitioned index, the index is split into components by partitioning the vocabulary, with one possible partition shown by the dotted horizontal split in Figure 9. Each processor has full information about a subset of the terms, meaning that to handle a query, only the relevant subset of the processors need to respond. Term partitioning has the advantage of requiring fewer disk seek and transfer operations during query evaluation than document-partitioning because each term's inverted list is still stored contiguously on a single machine rather than in fragments across multiple machines. On the other hand, those disk transfer operations each involve more data. More significantly, in a term-partitioned arrangement, the majority of the processing load falls to the coordinating machine, and experiments have shown that it can easily become a bottleneck and starve the other processors of work.

Choosing a Distribution Strategy. Document distribution typically results in a better balance of workload than does term partitioning and achieves superior query throughput. It also allows more naturally for index construction and for document insertion. On the other hand, index construction in a term-partitioned index involves first of all distributing the documents and building a document-partitioned index, and then exchanging index fragments between all pairs of processors after the vocabulary split has been negotiated. Document distribution also has the pragmatic advantage of still allowing a search service to be provided even when one of the hosts is offline for some reason since any answers not resident on that machine remain available to the system.

On the other hand, in a term-distributed system, having one machine offline is likely to be immediately noticeable.

The Google implementation uses a document-partitioned index with massive replication and redundancy at all levels: the machine, the processor cluster, and the site.

It is also worth noting that document partitioning remains effective even if the collaborating systems are independent and unable to exchange their index data. A distributed system in which a final result answer list is synthesized from the possibly-overlapping answer sets provided by a range of different services is called a *metasearcher*.

8. EFFICIENT INDEX REPRESENTATIONS

A central idea underpinning several aspects of efficient indexing and query processing is the use of compression. With appropriate compression techniques, it is possible to simultaneously reduce both space consumption and disk traffic. Compression also reduces overall query evaluation time. This section introduces a range of integer coding techniques, and describes their use in inverted file compression.

Compact Storage of Integers. In a simple implementation of an inverted file index, 32-bit and 16-bit integers might be used respectively for document identifiers and term frequencies since these sizes are universally supported by compilers and are large enough to hold the likely values. However, fixed-width integers are not particularly space-efficient, and large savings can be obtained using quite simple compression techniques.

An efficient representation for an inverted list requires a method for coding integers in a variable number of bits. Using a fixed number of bits for each number, whether 32 or 20, is inconvenient: it limits the maximum value that can be stored, and is wasteful if most numbers are small. Variable-length codes can be infinite, avoiding the problem of having a fixed upper bound; and can be constructed to favor small values at the expense of large.

Parameterless Codes. The simplest variable-bit infinite code is *unary* which represents the value $x > 0$ as $x - 1$ "1" bits followed by a terminating "0" bit.

Unary is an example of a fixed code corresponding to a fixed distribution and is unparameterized. Other unparameterized codes for integers are Elias' *gamma* and *delta* codes. In the gamma code, integer $x > 0$ is factored into $2^e + d$ where $e = \lfloor \log_2 x \rfloor$, and $0 \leq d < 2^e$. The codeword is formed as the concatenation of $e + 1$ represented in unary and d represented in binary in e bits. In the delta code, the value of $e + 1$ is represented using the gamma code and is followed by d in binary, as for gamma.

In both codes, each codeword has two parts, a prefix and a suffix. The prefix indicates the binary magnitude of the value and tells the decoder how many bits there are in the suffix part. The suffix indicates the value of the number within the corresponding binary range. Table II gives some example codewords for each of these three unparameterized codes. The colons used in the codewords to separate the prefix and suffix parts are purely illustrative.

Which code is preferable depends on the probability distribution $\Pr(x)$ governing the values x that are being coded. Using Shannon's relationship between the probability of a symbol and its ideal codeword length, $\text{len}(x) = -\log_2 \Pr(x)$, it can be seen that unary corresponds to the probability distribution $\Pr(x) = 2^{-x}$. That is, when half of all values are the number 1, a quarter are 2, an eighth are 3, and so on, then unary is the most efficient coding mechanism. When approximately half of all values are the number 1, a quarter are (equally one of) 2 or 3, an eighth are (equally one of) 4, 5, 6, or 7, and so on, and in general $\Pr(x) \approx 1/(2x^2)$, then gamma is the most efficient coding mechanism.

However, inspecting a sequence of bits one by one is relatively costly on machines for which the basic unit of access is multiples of eight bits. If each code occupies a sequence

Table II. Example Codewords Using the Unary, Gamma, and Delta Codes

value	unary	gamma	delta
1	0	0:	0::
2	10	10:0	10:0:0
3	110	10:1	10:0:1
4	1110	110:00	10:1:00
10	1111111110	1110:010	110:00:010
100		1111110:100100	110:11:100100
1,000		1111111110:111101000	1110:010:111101000

Colons are used as a guide to show the prefix and suffix components in each codeword. All three codes can represent arbitrarily large numbers; the unary codewords for 100 and 1,000 are omitted only because of their length.

To encode integer $x \geq 1$:

(1) Set $x \leftarrow x - 1$.
(2) While $x \geq 128$,
  (a) *write_byte*$(128 + x \bmod 128)$.
  (b) Set $x \leftarrow (x \text{ div } 128) - 1$.
(3) *write_byte*(x).

To decode an integer x:

(1) Set $b \leftarrow$ *read_byte*(), $x \leftarrow 0$, and $p \leftarrow 1$.
(2) While $b \geq 128$,
  (a) Set $x \leftarrow x + (b - 127) \times p$ and $p \leftarrow p \times 128$.
  (b) Set $b \leftarrow$ *read_byte*().
(3) Set $x \leftarrow x + (b + 1) \times p$.

Fig. 10. Encoding and decoding variable-length byte-aligned codes. Input values to the encoder must satisfy $x \geq 1$.

of whole bytes, the bit operations can be eliminated. With this in mind, another simple unparameterized code is to use a sequence of bytes to code a value $x \geq 1$, shown in Figure 10. The idea is very simple: if $x \leq 128$, then a single byte is used to represent $x - 1$ in binary, with a leading "0" bit; otherwise, the low-order seven bits of $x - 1$ are packed into a byte with a leading "1" bit, and the quantity (x div 128) is recursively coded the same way.

Thus the byte 2, with the bit-pattern 0000 0010, represents the integer 3; the byte 9, with the bit-pattern 0000 1001, represents the (decimal) integer 10; and the double-byte 147:7, with the bit-pattern 1110 0111 : 0000 0111, represents 1044 since $1044 = 128 \times (7 + 1) + (147 - 127)$. This approach is a form of gamma code in which the suffix length is a multiple of seven instead of a multiple of one. Several studies have explored bytewise coding and found it to be more efficient than bitwise alternatives. An enhanced version in which the split between continuing and terminating bytes is signaled by a different value than 128 has also been described and offers the possibility of improved compression effectiveness.

As well as offering economical decoding, the bytewise coding mechanism facilitates fast stepping through a compressed stream looking for the kth subsequent code. Since each codeword ends with a byte in which the top bit is a "0", it is easy to step through a compressed sequence and skip over exactly k coded integers without having to decode each of them in full.

To encode integer $x \geq 1$ using parameter b:

(1) Factor $x \geq 1$ into $q \cdot b + r + 1$ where $0 \leq r < b$.
(2) Code $q + 1$ in unary.
(3) Set $e \leftarrow \lceil \log_2 b \rceil$ and $g \leftarrow 2^e - b$.
(4) If $0 \leq r < g$ then code r in binary using $e - 1$ bits; otherwise, if $g \leq r < b$, then code $r + g$ in binary using e bits.

Fig. 11. Encoding using a Golomb code with parameter b. Input values must satisfy $x \geq 1$.

Table III. Example Codewords Using Three Different Golomb Codes

value	$b = 3$	$b = 5$	$b = 16$
1	0:0	0:00	0:0000
2	0:01	0:01	0:0001
3	0:11	0:10	0:0010
4	10:0	0:110	0:0010
10	1110:0	10:110	0:1001

Colons are used as a guide to show the prefix and suffix components in each codeword. All three codes can represent arbitrarily large numbers.

Golomb and Rice Codes. The Elias codes just described have the property that the integer 1 is always encoded in one bit. However, d-gaps of 1 are not particularly common in inverted lists. More pertinently, within the context of a single inverted list with known length, the likely size of the numbers can be accurately estimated. It is therefore desirable to consider schemes that can be parameterized in terms of the average size of the values to be coded.

Golomb codes, described in Figure 11, have this property. Integer $x \geq 1$ is coded in two parts—a unary bucket selector, and a binary offset within that bucket. The difference between this and the Elias codes is that, in Golomb codes, all buckets are the same size. The use of a variable-length binary code for the remainder r means that no bit combinations are wasted even when the parameter b is not an exact power of two. Table III gives some examples of the codewords generated for different values of b. When b is a power of two, $b = 2^k$, the suffix part always contains exactly k bits. These are known as Rice codes and allow simpler encoding and decoding procedures. An example Rice code appears in the last column of Table III.

Matching Code to Distribution. The implied probability distribution associated with the Elias gamma code was described earlier; Golomb codes are similarly minimum-redundancy when

$$\Pr(x) \approx (1 - p)^{x-1} p\,,$$

provided that

$$b = \left\lceil \frac{\log(2 - p)}{-\log(1 - p)} \right\rceil \approx 0.69 \times \frac{1}{p}\,,$$

where p is the parameter of the geometric distribution (the probability of success in a sequence of independent trials). As we now explain, an inverted list can be represented as a sequence of integers that are, under a reasonable assumption, a sequence of independent (or Bernoulli) trials.

Binary Codes. Other coding mechanisms can also perform well and even binary codes can realize compact representations if the data is locally homogeneous. At the same time, they can provide decoding speeds as good as bytewise codes. In the simplest

example of this approach, a sequence of values is represented by fitting as many binary codes as possible into the next 32-bit output word. For example, if all of the next seven sequence values are smaller than 17, then a set of 4-bit codewords can be packed into a single output word. Similarly, if all of the next nine values are smaller than 9, an output word containing 3-bit codewords can be constructed. To allow decoding, each word of packed codes is prefixed by a short binary selector that indicates how that word should be interpreted.

As with the bytewise coding method, this approach supports fast identification of the kth subsequent codeword since all that is required is that the selector of each word be examined to know how many codewords it contains.

Compressing Inverted Lists. The representation of inverted lists introduced earlier described each inverted list as a sequence of $\langle d, f_{d,t} \rangle$ values with the restriction that each document identifier d be an ordinal number. From a database perspective, imposing an ordinal identifier appears to be a poor design decision because it has implications for maintenance. However, text databases are not manipulated and updated in the ways that relational databases are, and, in practice, the use of a simple mapping table obviates the difficulties.

The frequencies $f_{d,t}$ are usually small (with a typical median of 1 or 2) and can be efficiently represented using unary or gamma. But representing raw document identifiers using these codes gives no saving since the median value in each inverted list is $N/2$.

On the other hand, if document identifiers are sorted and first-order differences (d-*gaps*) are stored, significant savings are possible. If a term appears in a random subset of f_t of the N documents in the collection, the d-gaps conform to a geometric distribution with probability parameter $p = f_t/N$ and can thus be coded effectively using a Golomb code or one of the other codes. For example, a Golomb-gamma-coded index of all words and numbers occurring in the documents for the `NewsWire` collection would occupy about 7% of the text size. In this approach, the representation for each inverted list consists of alternating d-gap values and $f_{d,t}$ values, each occupying a variable number of bits. The d-gap values are represented as Golomb codes, using a parameter b determined from the number of $\langle d, f_{d,t} \rangle$ pairs in that inverted list, and the $f_{d,t}$ values are represented using the gamma code.

Use of Golomb codes does, however, present problems for the update strategy of intermittent merge. If Golomb codes are used for representing document identifiers, the merge process involves decoding the existing list and recoding with new parameters, but processing the existing list will then be the dominant cost of update, and should be avoided. One solution is that the old Golomb parameter could continue to be used to compress the added material at some small risk of gradual compression degradation. Alternatively, if static codes are used and lists are document-ordered, new $\langle d, f_{d,t} \rangle$ values can be directly appended to the end of the inverted lists. For other list orderings, such as those described in Section 9, there may be no alternative to complete reconstruction.

Compression Effectiveness. The effectiveness of compression regimes is particularly evident for inverted lists representing common words. To take an extreme case, the word `the` is likely to occur in almost every document (in an English-language collection), and the vast majority of d-gaps in its inverted list will be 1 and represented in just a single bit. Allowing for the corresponding $f_{d,t}$ value to be stored in perhaps (at most) 10–11 bits, around 12 bits is required per $\langle d, f_{d,t} \rangle$ pointer for common words, or one-quarter of the 48 bits that would be required if the pointer was stored uncompressed. The additional space savings that can be achieved with stopping are, for this reason, small.

However, stopping does yield significant advantages during index maintenance because stopping means that updates to the longest lists (those of common words) are avoided.

Less frequent words require longer codes for their d-gaps but, in general, shorter codes for their $f_{d,t}$ values. As a general rule-of-thumb, each $(d, f_{d,t})$ pointer in a complete document-level inverted index requires about 8 bits when compressed using a combination of Golomb and Elias codes. Use of the less-precise bytewise code for one of the two components adds around 4 bits per pointer and, for both components, adds around 8 bits per pointer. The word-aligned code has similar performance to the bytewise code for large d-gaps but obtains better compression when the d-gaps are small.

For example, a bytewise-gamma-coded index for the `NewsWire` data (with the d-gaps represented in a bytewise code and the $f_{d,t}$ values coded using gamma) would occupy about $70 \times 10^6 \times 12$ bits or approximately 10% of the source collection. In this index, the document identifiers are byte aligned, but the $f_{d,t}$ values are not, so the list is organized as a vector of document numbers and a separate vector of frequencies with the two lists decoded in parallel when both components are required. While the compression savings are not as great as those attained by Golomb codes, the bytewise and word-aligned codes bring other benefits.

Compression of Word Positions. Uncompressed, at, for instance, two bytes each (so no document can exceed 65,536 words), word positions immediately account for the bulk of index size: 360MB for `NewsWire`, and 22GB for `Web`. However, these costs can be reduced by taking differences, just as for document identifiers, and Golomb codes can be used to represent the differences in either a localized within-this-document sense, or, more usefully, in an average-across-all-documents manner. In the latter case, the vocabulary stores two b values, one used to code the d-gaps in the way that was described earlier, and a second to code the w-gaps counting the word intervals between appearances of that term. Static codes—gamma, delta, bytewise, or word-aligned—also give acceptable compression efficiency and may be attractive because of their simplicity.

The choice of coding scheme also affects total fetch-and-decode times with the bytewise and word-aligned codes enjoying a clear advantage in this regard. Bitwise codes cannot be easily stepped through, and queries that require only bag-of-words processing can be considerably slower with an interleaved word-level index than with a document-level index.

An aspect of inverted indexes that is dramatically altered by the introduction of word positions is the cost of processing common words. In a document-level index, common words such as `the` are relatively cheap to store. In a word-level index, the average per-document requirement for common words is much larger because of the comparatively large number of $f_{d,t}$ word-gap codes that must be stored. In the case of the `Web` data, just a handful of inverted lists account for more than 10% of the total index size. Processing (or even storage) of such lists should be avoided whenever possible.

Nonrandom Term Appearances. If documents are chronological, terms tend to cluster. For example, in the 242,918 documents of the Associated Press component of TREC, there are two definite clusters for `hurricane`, one due to hurricane Gilbert in September 1988, and a second resulting from hurricane Hugo in September 1989. Similar clustering arises with web crawls. Some sites are topic-specific so that words that are rare overall may be common within a group of pages. A particular cause of this effect is indexing of documents that are written in a variety of languages.

The nonuniform distribution can be exploited by an *interpolative code* which transmits the mid-point of the list of document numbers in binary, then recursively handles the two sublists in the resulting narrowed ranges. For typical collections, this code

results in an average of 0.5 to 1.5 bits saved per $\langle d, f_{d,t} \rangle$ pointer compared to a Golomb code. The drawback of the interpolative code is that it is more complex to implement and slower in decoding.

The word-aligned binary coding method is also sensitive to localized clustering, a run of consistently small d-gaps is packed more tightly into output words than is a sequence containing occasional larger ones.

Pros and Cons of Compression. Compression has immediately obvious benefits. It reduces the disk space needed to store the index; and during query evaluation, it reduces transfer costs (the lists are shorter) and seek times (the index is smaller). Compression also reduces the costs of index construction and maintenance.

A less obvious benefit of index compression is that it improves caching. If a typical inverted list is compressed by a factor of six, then the number of inverted lists that can be retained in memory is increased by that same factor. In a retrieval system, queries arrive with a skew distribution with some queries and query terms much more common than others. Queries for `Chicago weather forecast` far exceed those for `Kalgoolie weather forecast`, for example. There can also be marked temporal effects. People use Web search engines to check questions on broadcast quiz shows, for example, so the same unusual query might arrive thousands of times in a short space of time. Increasing the effectiveness of caching can dramatically cut the cost of resolving a stream of queries.

The principal disadvantage of compression is that inverted lists must be decoded before they are used. A related problem is that they may need to be recoded when they are updated, for example, if parameters change. For some of the codes considered, the addition of new information can require complex recoding.

On current desktop machines, the decoding cost is more than offset by the reduction in disk seek costs. More importantly, as the ratio of the speed between processors and disk continues to diverge, the performance benefit available through compression is increasing. For byte- and word-aligned codes, which allow each document identifier to be decoded in just a few instruction cycles, the processing cost is more than offset by the reduction in memory-to-cache transfer costs.

Thus, if the index is larger than the available main memory or cannot be buffered for some other reason, there is no disadvantage to compression. And even if the index is in memory, processing can be faster than for uncompressed data. Use of appropriate index compression techniques is an important facet of the design of an efficient retrieval system.

9. LIMITING MEMORY REQUIREMENTS

If the standard query evaluation algorithm is used for queries that involve common words, most accumulators are nonzero, and an array of N entries A_d is the most space- and time-efficient structure. But the majority of those accumulator values are trivially small, as the only matching terms are one or more common words. Analysis of search engine logs has demonstrated that common terms are not the norm in queries, and analysis of relevance has demonstrated that common terms are of low importance. It thus makes sense to ask if there are better structures for the accumulators, requiring fewer than N elements.

Accumulator Limiting. If only documents with low f_t (i.e., rare) query terms are allowed to have an accumulator, the number of accumulators can be greatly reduced. This strategy is most readily implemented by imposing a limit L on the number of accumulators as shown in Figure 12.

To use an inverted index and an accumulator limit L to rank a document collection with regard to a query q and identify the top r matching documents:

(1) Create an empty set A of accumulators.
(2) For each query term t in q, ordered by decreasing $w_{q,t}$,
  (a) Fetch the inverted list for t.
  (b) For each pair $\langle d, f_{d,t} \rangle$ in the inverted list
    i. If A_d does not exist and $|A| < L$, create accumulator A_d.
    ii. If A_d exists, calculate $w_{d,t}$ and set $A_d \leftarrow A_d + w_{q,t} \times w_{d,t}$.
(3) Read the array of W_d values.
(4) For each accumulator $A_d \in A$, set $S_d \leftarrow A_d / W_d$.
(5) Identify the r greatest S_d values and return the corresponding documents.

Fig. 12. The limiting method for restricting the number of accumulators during ranked query evaluation. The accumulator limit L must be set in advance. The thresholding method involves a similar computation, but with a different test at step 2(b)i.

In ranking, using the models described earlier, a document can have a high similarity score even if several query terms are missing. In the limiting approach, as each $\langle d, f_{d,t} \rangle$ value is processed, a check is made to determine whether there is space in the set of accumulators to consider additional documents. Thus the algorithm enforces the presence of the query terms that are weighted highly because of their frequency of occurrence in the query or because of their rareness in the collection. While the behavior of the ranking is altered, it is arguably for the better since a document that is missing the most selective of the query terms is less likely to be relevant. Some of the Web search engines only return matches in which all of the query terms are present, a strategy that appears to be effective in practice for short queries but, in experimental settings, has been found to be less compelling.

Accumulator Thresholding. Another approach of a similar type is to use partial similarities—the contribution made by a term to a document's similarity, or $w_{q,t} \times w_{d,t}$—to determine whether an accumulator should be created. In this approach, accumulators are created if it seems likely that the document will ultimately have a sufficiently high similarity, with the test $|A| < L$ in Figure 12 replaced by a test $w_{q,t} \times w_{d,t} > S$, for some value S. The threshold S is initially small but is increased during query processing so that it becomes harder for documents to create an accumulator as the computation proceeds.

For a set of accumulators that is limited to approximately 1%–5% of the number of documents, in the context of TREC-style long queries and evaluation, there is no negative impact on retrieval effectiveness.

However, with Web-style queries, both the limiting and the thresholding methods can be poor. In the limiting method, the limit tends to be reached part way through the processing of a query term, and thus the method favors documents with low ordinal identifiers; but if the limit is only tested between lists, then the memory requirements are highly unpredictable. Similar problems affect thresholding. A solution is to modify the thresholding approach so that, not only are accumulators created when the new contribution is significantly large, but existing accumulators are discarded when they are too small. With this approach, the number of accumulators can be fixed at less than 1% of the number of documents for large Web-like collections.

Data Structures. Comparing the two approaches, the limit method gives precise control over memory usage, whereas the threshold method is less rigid and allows even common terms to create accumulators for particular documents if they are sufficiently

frequent in those documents. Both methods reduce the number of nonzero accumulators, saving memory space without affecting retrieval effectiveness. And, as examples of more general *query pruning methods*, they can also reduce disk traffic and CPU time, using methods that are discussed shortly.

To maintain the set of accumulators in either of these two approaches, a data structure is required. An obvious solution is to store the accumulators as a list ordered by document number and successively merge it with each term's inverted list.

Representing Document Lengths. Storage of the set of document lengths W_d is another demand on memory during ranking. Figures 2, 4, and 12 suggest that the W_d values be stored in a file on disk, but execution is faster if they can be retained in memory.

There are two aspects of their use that make memory residence possible. First, they are an attribute of the document collection rather than of any particular query. This means that the array of W values can be initialized at system startup time rather than for every query and that, in a multithreaded evaluation system, all active queries can access the same shared array.

The second aspect that allows memory-residence is that, like many other numeric quantities associated with ranked query evaluation, they are imprecise numbers and result from a heuristic rather than an exact process. Representing them to 64 or 32 bits of precision is, therefore, unnecessary. Experiments have shown that use of 8- or 16-bit approximate weights has negligible effect on retrieval effectiveness.

Storage for the Vocabulary. Except for the document mapping table and assorted buffers, the only remaining demand on main memory is the term lookup dictionary or vocabulary. For collections such as `NewsWire` and `Web`, retaining the vocabulary in main memory is expensive since each entry includes a word, ancillary fields such as the weight of that term, and the address of the corresponding inverted list.

Fortunately, access to the vocabulary is only a small component of query processing; if a B-tree-like structure is used with the leaf nodes on disk and internal nodes in memory, then a term's information can be accessed using just a single disk access, and only a relatively small amount of main memory is permanently consumed. For example, consider the `Web` collection described in Table I. The 16 million distinct terms in that collection correspond to a vocabulary file of more than 320MB but if each leaf node contains (say) 8kB of data, around 400 vocabulary entries, then the in-memory part of the structure contains only 40,000 words, occupying less than 1MB. As for the document weights, this memory is shared between active queries and is initialized at startup rather than once per query.

10. REDUCING RETRIEVAL COSTS

The core implementation, together with simple strategies for reducing memory consumption, provides a functional query evaluation system. For example, in the context of a desktop machine and a gigabyte-scale text collection, typical queries of a few terms can be resolved using these techniques in well under a second. In addition, with accumulator limiting, compact document weights, and a B-tree for the vocabulary, the memory footprint can be limited to a dozen or so megabytes. However, substantial further savings can be made, and, with careful attention to detail, a single standard PC can easily provide text search for a large organization such as a university or corporation.

In the core implementation (Figure 4), the principal cost is retrieval and decoding of inverted lists. That is, every $\langle d, f_{d,t} \rangle$ pair contributes to a valid accumulator. With limited-accumulator query evaluation (Figure 12), however, most $\langle d, f_{d,t} \rangle$ pairs do not correspond to a valid accumulator, and processing time spent decoding these pairs is wasted. We also argued earlier that to facilitate compression and the gains that

compression brings, the document numbers in each list should be sorted. However, use of compression means that each number is stored in a variable number of bits or bytes so random access (e.g., to support binary search) into inverted lists is not possible.

Skipping. A proposal from the mid-1990s is that decoding costs be reduced by the insertion of additional structure into inverted lists. In this *skipping* approach, descriptors are periodically embedded in each compressed inverted list, dividing the list into chunks. The descriptor can be used to determine whether any $\langle d, f_{d,t} \rangle$ value in the chunk corresponds to a valid accumulator; if not, that chunk does not need to be decoded, and processing can immediately move to the next chunk. However, all chunks still need to be fetched. Note also that, if the chunks are as large as disk blocks, the great majority will need to be decoded, and, if they are smaller, more chunks can be skipped but most disk blocks will still contain a valid chunk. In experiments at the time, we found that skipping yielded worthwhile performance gains. Since then, processors have become much faster, while disk access times have not greatly improved, and the benefits provided by skipping have eroded.

To reduce disk transfer costs, it is necessary to avoid fetching the inverted lists in their entirety. One obvious method is to only fetch inverted lists for rare terms, but experiments consistently show that all terms should be processed if effectiveness is to be maintained. The more attractive option is to rearrange inverted lists so that, in a typical query, only part of each relevant list need be fetched. The schemes described earlier for accumulator limitation provide criteria for deciding whether a $\langle d, f_{d,t} \rangle$ value will be used; we now explain how these criteria can be used to restructure inverted lists without seriously degrading compression effectiveness.

Frequency-Ordered Inverted Lists. The principle of accumulator limitation is that large values of $w_{q,t} \times w_{d,t}$ should be processed first. The default heuristic for identifying these values is to process the low-f_t terms first as their lists are likely to contain more large $w_{q,t} \times w_{d,t}$ values than would the lists of high-f_t terms. However, in a typical inverted list, most $f_{d,t}$ values are small, while only a few are large. If only the large $f_{d,t}$ values are interesting, that is, can contribute to a useful $w_{q,t} \times w_{d,t}$ value, then they should be stored at the beginning of their inverted list rather than somewhere in the middle of the document-based ordering. For a given threshold S and term t, all $w_{q,t} \times w_{d,t} > S$ values (see Equation (1)) will then be stored before any smaller ones.

To make use of this idea, the index can be reorganized. A standard inverted list is sorted by document number, for example,

$\langle 12, 2 \rangle$ $\langle 17, 2 \rangle$ $\langle 29, 1 \rangle$ $\langle 32, 1 \rangle$ $\langle 40, 6 \rangle$ $\langle 78, 1 \rangle$ $\langle 101, 3 \rangle$ $\langle 106, 1 \rangle$.

When the list is reordered by $f_{d,t}$, the example list is transformed into

$\langle 40, 6 \rangle$ $\langle 101, 3 \rangle$ $\langle 12, 2 \rangle$ $\langle 17, 2 \rangle$ $\langle 29, 1 \rangle$ $\langle 32, 1 \rangle$ $\langle 78, 1 \rangle$ $\langle 106, 1 \rangle$.

The repeated frequency information can then be factored out into a prefix component with a counter inserted to indicate how many documents there are with this $f_{d,t}$ value:

$\langle 6 : 1 : 40 \rangle$ $\langle 3 : 1 : 101 \rangle$ $\langle 2 : 2 : 12, 17 \rangle$ $\langle 1 : 4 : 29, 32, 78, 106 \rangle$.

Finally, if differences are taken in order to allow compression, we get

$\langle 6 : 1 : 40 \rangle$ $\langle 3 : 1 : 101 \rangle$ $\langle 2 : 2 : 12, 5 \rangle$ $\langle 1 : 4 : 29, 3, 46, 28 \rangle$.

Repeated frequencies $f_{d,t}$ are not stored, giving a considerable saving. But within each equal-frequency segment of the list, the d-gaps are now on average larger when the document identifiers are sorted, and so the document number part of each pointer

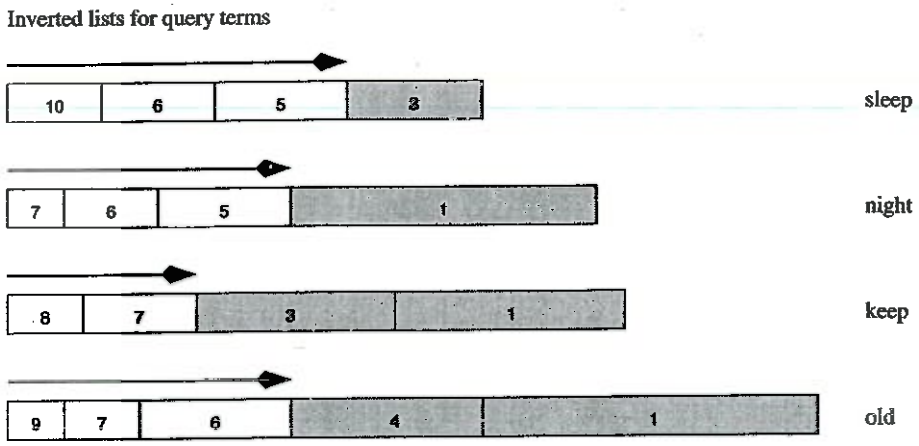

Fig. 13. Interleaved processing of inverted lists using a frequency- or impact-sorted index. Blocks from the front of each list are applied to the set of accumulators in decreasing score order until some stopping condition has been met. At the instant shown, 11 blocks have been processed, and, if processing continues, the next block to be processed would be for the term `old`.

increases in cost. In combination, these two effects typically result in frequency-sorted indexes that are slightly smaller than document-sorted indexes.

Using a frequency-sorted index, a simple query evaluation algorithm is to fetch each list in turn, processing $\langle d, f_{d,t} \rangle$ values only while $w_{q,t} \times w_{d,t} \geq S$, where S is the threshold. If disk reads are performed one disk block at a time rather than on a whole-of-inverted-list basis, this strategy significantly reduces the volume of index data to be fetched without degrading effectiveness.

A practical alternative is for the first disk block of each list to be used to hold the $\langle d, f_{d,t} \rangle$ values with high $f_{d,t}$. These important blocks could then all be processed before the remainder of any lists, ensuring that all terms are given the opportunity to create accumulators. The remainder of the list is then stored in document order.

A more principled method of achieving the same aim is to interleave the processing of the inverted lists. In particular, once the first block of each list has been fetched and is available in memory, the list with the highest value of $w_{q,t} \times w_{d,t}$ can be selected, and its first run of pointers processed. Attention then switches to the list with the next-highest $w_{q,t} \times w_{d,t}$ run which might be in a different list or might be in the same list.

During query processing, each list could be visited zero, one, or multiple times, depending only on the perceived contribution of that term to the query. The interleaved processing strategy also raises the possibility that query evaluation can be terminated by a time-bound rather than a threshold since the most significant index information is processed first. Figure 13 shows the interleaved evaluation strategy with list sections presumed to be applied to the set of accumulators in decreasing order of numeric score. The shaded sections of each list are never applied.

Impact-Ordered Inverted Lists. An issue that remains even in the frequency-sorted approach is that $w_{q,t} \times w_{d,t}/W_d$ is the true contribution of term t to the similarity score $S_{q,d}$, rather than $w_{q,t} \times w_{d,t}$. If it makes sense to frequency-sort the inverted lists into decreasing $w_{d,t}$ order, then it makes even better sense to order them in decreasing *impact* order using $w_{d,t}/W_d$ as a sort key. Then all that remains is to multiply each stored value by $w_{q,t}$ and add it to the corresponding accumulator. That is, an impact value incorporates the division by W_d into the index.

To use an impact-sorted inverted index and an accumulator limit L to rank a document collection with regard to a query q and identify the top r matching documents:

(1) Create an empty set A of accumulators.
(2) Fetch the first block of each term t's inverted list. Let s_t be the stored impact score for that block.
(3) While processing time is not exhausted and while inverted list blocks remain
  (a) Identify the inverted list block of highest $w_{q,t} \times s_t$, breaking ties arbitrarily. Let C be the integer contribution derived from $w_{q,t} \times s_t$.
  (b) For each document d referenced in that block,
    i. If A_d does not exist and $|A| < L$, create an accumulator A_d.
    ii. If A_d exists, set $A_d \leftarrow A_d + C$.
  (c) Ensure that the next equi-impact block for term t is available, and update s_t.
(4) Identify the r greatest A_d values and return the corresponding documents.

Fig. 14. Impact-ordered query evaluation.

If the inverted lists are then sorted by decreasing impact, the same interleaved processing strategy can be employed. Now the blocks in the inverted list must be created artificially rather than occurring naturally, since $w_{d,t}/W_d$ is not integer-valued, and exact sorting would destroy the d-gap compression. Hence, to retain compression, the impact values are quantized so that each stored value is one of a small number of distinct values, and integer surrogates are stored in the inverted lists in place of exact $w_{d,t}/W_d$ values. Experiments with this technique have shown that approximately 10–30 distinct values suffice to give unaltered retrieval effectiveness compared to full evaluation. And, unlike frequency-based methods, impact-sorted lists are just as efficient for evaluation of the Okapi similarity heuristic described in Equation (2).

With this change, each list is a document-sorted sequence of blocks of approximately-equal-impact pointers, and no $f_{d,t}$ values are stored. Compared to document- and frequency-sorted indexes, the compressed size grows slightly because the average d-gaps are bigger. Nevertheless, the index is still a small fraction of the size of the text being indexed.

Query evaluation with an impact-ordered index becomes a matter of processing as many blocks as can be handled in the time that is available. To process a block, an integer contribution score that incorporates the impact score associated with that block and the query term weight $w_{q,t}$ (which might be unity) is added to the accumulator of every document recorded in the block. All query-time operations are on integers, also making the final extraction phase significantly easier to manage. Impact-ordered query evaluation is shown in Figure 14.

With an impact-sorted inverted file and integer surrogate weights, query processing is extremely fast. Just as important is that time consumed by each query can be bounded, independent of the number of terms in the query. When query load is light or when a premium service has been purchased, a greater fraction of the blocks in each query can be used to influence the result ranking.

Impacts provide another illustration of the flexible constraints on design of algorithms for information retrieval. Information can be selectively thrown away or approximated without detriment to the output of the algorithm, although there may be a beneficial change to efficiency. While the set of documents returned by impact-based processing will not be the same as the set returned by the underlying computation in Equation (1), the quality of the set (proportion of documents that are relevant) may be unchanged. In practice, the effect is likely to be that one method gives better results

than the other for some queries and worse results than the other for other queries. Such change-but-no-difference effects are commonly observed in development of algorithms for text query evaluation.

Other Considerations. When the inverted lists are impact- or frequency-sorted, they are read in blocks rather than in their entirety, and contiguous storage is no longer a necessity. Blocks with high-impact information could be clustered on disk, further accelerating query processing. The lists for common terms—the ones with many document pointers—will never be fully read, and a great deal of disk traffic is avoided. Experiments with these techniques, using collections such as `NewsWire`, show that disk traffic and processing time are reduced by a factor of up to three compared to previous techniques involving limited accumulators. The likely gain is even greater for the larger `Web` collection. As a final advantage, the impact-sorted index dispenses with the need for an array or file of W_d values.

The disadvantage of both frequency- and impact-sorted indexes is that Boolean queries and index updates are more complex. To handle Boolean queries efficiently, other index structures have been proposed. One approach is to use a *blocked* index. Rather than use d-gaps that are relative to the previous document number, it is feasible to code d-gaps relative to the start of a block. This allows use of a form of binary search at some loss of compression efficiency. Conjunctive Boolean queries can then be processed extremely quickly.

However, updates remain complex. With document-ordered list organizations, new $\langle d, f_{d,t} \rangle$ values can be appended to the list. In the enhanced organizations, the whole of each list must be fetched and decoded. Whether this is a significant disadvantage depends on the application.

Summary. We have described compression-based indexing and query evaluation techniques for bag-of-word queries on a text database. Compared to the uncompressed, document-sorted implementation described in Figure 4,

—*disk space* for a document-level index requires about 7%–9% of the size of the data and has been reduced by a factor of around five;

—*memory space* for accumulators requires a dozen bytes for 5% of the documents in a small collection or less than 1% of the documents in a large collection and has been reduced by a factor of around twenty;

—*CPU time* for processing inverted lists and updating accumulators is reduced by a factor of three or more;

—*disk traffic* to fetch inverted lists is reduced by a factor of five in volume and of two or more in time (The relative savings increase with collection size);

—*throughput* of queries is reduced by making better use of memory, allowing more effective caching of vocabulary entries, lists, and answers;

—*construction time* for indexes is cut by a factor of more than two, through reduction in the disk-traffic bottleneck and reduction in the number of runs required to build an index.

11. OTHER APPROACHES TO INDEXING

Querying can also be carried out without an index. For typical data and queries, building an index requires about 10–50 times the cost of a string search. If queries are rare, or the data is extremely volatile, it is reasonable to use a document-oriented querying strategy of the type shown in Figure 2. An example is searching of a personal email directory or search within a small collection of files.

A. *Original string, indexed by ordinal position:*

```
0         1         2         3         4
01234567890123456789012345678901234567890123456
The old night keeper keeps the keep in the town
```

B. *Selected character-pairs in the original string, indexed by ordinal position:*

```
00   04   08   14   21   27   31   36   39   43
Th.. ol.. ni.. ke.. ke.. th.. ke.. in.. th.. to..
```

C. *The same character-pairs, after lexicographic sorting:*

```
36   31   14   21   08   04   27   00   39   43
in.. ke.. ke.. ke.. ni.. ol.. th.. Th.. th.. to..
```

Fig. 15. Example of a suffix array.

For querying of larger collections, inverted files are not the only technology that has been proposed. The two principal alternatives are suffix arrays and signature files. In this section, we briefly outline these approaches and explain why they are not competitive with inverted files for the applications considered in this article.

Suffix Arrays. A suffix array stores pointers into a source string, ordered according to the strings they point at. The index can be of every character position or, for typical retrieval applications, every word position. The structure provides access via a binary search-like process which is fast if it can be assumed that both the array and the underlying source string are available in memory.

Consider the first line of the Keeper collection annotated with byte offsets in the string, commencing at zero, as shown in part A of Figure 15. In a word-level suffix array, a list is made of the byte addresses at which words start, shown in the first row of part B of Figure 15. The second row of part B shows the initial two characters of the semi-infinite string that is the target of each pointer. To form the suffix array, the byte pointers are reordered so that the underlying strings are sorted, using as many characters of the semi-infinite strings as are necessary to break ties, as shown in part C. Note that the second of these two lines is provided only to allow the destinations of the pointers to be visualized by the reader. All that is actually stored is the original string, and an array `A` containing one pointer for each word in the string, with `A[0]=36`, `A[1]=31`, and so on.

One way of interpreting the resultant array of byte pointers is that they represent a combined vocabulary and inverted index of every suffix string in the original text with access provided directly to the bytes rather than via ordinal document identifiers. To use the suffix array, the sorted byte pointers are binary searched, again using the underlying strings to drive the comparisons. The output of each search is a pair of indices into the suffix array, describing the first and last locations whose strings are consistent with the supplied pattern. For example, to locate occurrences of keep in the example array, the binary search would return 1 and 3 since all of `A[1]`, `A[2]`, and `A[3]` contain byte addresses (31, 14, and 21, respectively) pointing at semi-infinite strings in the original text that commence with the prefix keep. (If exactly the word keep was required rather than as a prefix, a trailing delimiter could be added to the search key, and then only `A[1]=31` would be returned.)

It is straightforward to extend suffix arrays to phrase searching, and a range of simple techniques, such as conflating all nonalphabetic characters to a single separator character for the purposes of string comparison, allow the same binary-searching process to be applied regardless of the length or frequency of the phrase or words in it.

Case-folding can also be achieved by making the string comparison function case-insensitive as has been presumed in the example.

For large-scale applications, suffix arrays have significant drawbacks. The pointer array is accessed via binary search so compression is not an option. For a word-aligned suffix array, a 4-byte pointer is needed for each 6 bytes or so of text, and the underlying text must also be retained. In total, the indexing system requires around 170% of the space required by the input, all of it memory-resident. A suffix array-based retrieval system for 1GB of text demands a computer with 2GB or more of memory, a requirement that scales linearly as the source message grows. Suffix array indexes are large compared to inverted files because the repeated information cannot be usefully factored out and because byte addresses are used rather than word-offset addresses. If the base text is large, the suffix array itself can be indexed and partial strings stored in index to avoid needing to access the base text. For very large texts, the pointers can be managed hierarchically. However, these techniques add considerable complexity to the building and searching processes.

Another drawback is that there is no equivalent of ranked querying. All but simple stemming regimes are also problematic. On the other hand, suffix arrays offer increased string searching functionality compared to inverted files—complex patterns, such as wild-characters, can be handled. The principal strength of suffix arrays is that they greatly accelerate grep-style pattern matching, swapping decreased time for increased space.

Worth noting is that much of the functionality offered by a suffix array can be achieved in the context of an inverted file by indexing the vocabulary either with a suffix array spanning the vocabulary strings themselves or by a secondary inverted index making use of the character bigrams or trigrams that comprise the vocabulary terms.

Signature Files. Signature files are a probabilistic indexing method. For each term, s bits are set in a *signature* of w bits. The term descriptors for the terms that appear in each document are superimposed (i.e., OR'ed together) to obtain a document descriptor, and the document descriptors are then stored in a *signature file* of size wN bits for N documents.

To query on a term, the query descriptor for that term is formed; all of the documents whose document descriptors are a superset of the query descriptor are fetched; and the *false matches* are separated from the *true matches* with only the latter returned to the user as answers.

Assuming that all records are the same length, that is, then contain the same number of distinct terms; all terms occurring in over 5% of records are stopped; and there is on average just one false match per single-term query, then the index size is about 20% of the text. With an unstopped index, signatures must be much wider, giving a 40%-of-text index.

Variability in document length significantly complicates these calculations. With a fixed-length signature, more bits are set for long documents than short. These documents are then more likely to be selected as false matches, and because they are long, the cost of false-match checking is further increased. To get a false-match volume of one average document-length, a signature file needs to be much larger than the indicative sizes given here.

The simplest organization for a signature file is a sequence of signatures, which implies that the whole index is processed in response to each query. A range of alternative organizations of two kinds have been proposed. One is bitslicing in which the signature file, viewed as a matrix, is transposed in which case it is only necessary to fetch a small number of slices—perhaps 10 to 15—in response to a query. Each slice has one bit per stored record, and so, for the Web collection, would contain 1.5MB of data.

Another alternative organization is the use of descriptors in which signatures are partitioned according to a small number of selected bits and only matching partitions are fetched for each query. We are unaware of any practical demonstration of the merits of such descriptor schemes. Bitslicing, however, has been explored experimentally, including a range of variations designed to reduce costs. For example, slice length can be dramatically reduced with a small increase in the number of slices fetched, although costs remain linear in the number of stored documents.

Even for the relatively unsophisticated task of Boolean retrieval, signature files have several crucial defects. One is the need to eliminate false matches. For a given signature width, the number of false matches is linear in the collection size and hence, as the number of indexed documents grows, the number of documents unnecessarily fetched increases. For any reasonable parameter settings, this cost must dominate. And the longer a document, the higher the likelihood that it will be retrieved as a false match. Second, signature file indexes are large compared to compressed inverted files and are no easier to construct or maintain. Third, they require more disk accesses for short queries—exactly the kind that is commonplace in Web searching. And fourth, they are even more inefficient if Boolean OR and NOT operators are permitted in addition to AND.

More critically, there is no sensible way of using signature files for handling ranked queries or for identifying phrases. That is, there are no situations in which signature files are the method of choice for text indexing even for exact-match Boolean style searching.

12. BEYOND AN ELEMENTARY SEARCH ENGINE

We have outlined indexing and query evaluation algorithms that can be used in a practical search engine. These algorithms can be used to index large volumes of data and provide rapid response to user queries. However, in specific searching applications, further improvements in performance are available. In this section, we briefly review some of these options.

Crawling. Data acquisition is a key activity in dynamic systems. In unusual cases, the revised content will be delivered as it is generated, and we need do no more than index it. For example, a commercial arrangement might result in the daily delivery of updated catalog entries for an online store.

More usually, however, the owners of data change it without regard for whether or not it is indexed. *Crawling* is the process of seeking out changed data and feeding it back to the retrieval system for incorporation into updated indexes. In principle, crawling is easy, a set of seed URLs is used as the basis for the search, and those pages are fetched one by one. Each hyperlink within those pages is extracted and, if it has not been explored yet, appended to a queue of pending pages. Eventually, all pages reachable from the seed set will have been accessed, and the process can be repeated.

However, there are many subtle issues that require attention if a crawler is to be efficient. First, not all pages are of the same importance, and, while crawling the CNN Web site at hourly intervals might be appropriate, crawling a set of University pages probably is not. Indeed, the volume of data available might mean that the crawl simply does not finish before it is necessary to start it again. In this case, a strategy for prioritizing page visits and aborting the crawl when only low-priority pages remain may be required.

Second, not all pages are edited at the same rate, and editing may follow a regular pattern. An adaptive crawler might be able to significantly reduce crawl volume by identifying the update strategy that applies to pages and learning which pages are almost never modified.

Third, there are crawling traps that need to be anticipated and avoided—unintentional traps such as recursive script-based Web pages that implement calendars with a "next month" link are just one example. Fourth, the crawler needs to be sensitive to the requirements of the site it is visiting and must stagger its requests so as to avoid flooding it over a short period of time. Fifth, the crawler must be alert to the problem of duplicate pages and mirrored sites. And, finally, crawling is expensive because it involves either payment per gigabyte of data fetched (for some connection modes) whether it is useful or not, or paying for a high-capacity connection to the Internet so that the necessary bandwidth can be achieved.

Caching. In many contexts, the same queries and query terms recur. For example, many queries to the Web search engines are topical, and queries to a site-specific search engine may also be clustered. (Many of the queries on the Web site of one of our universities contain the query term `results`. Over half of these queries are posed on the two single days after first and second semester on which student grades are released.) A search engine can take advantage of this behavior by caching.

There are several kinds of information that can be cached. The inverted lists that are fetched in response to a query can be explicitly kept in memory. Alternatively, if these lists are maintained in disk blocks, the operating system is likely to cache them automatically on the basis of most-recently-used. As was noted earlier, compression helps by increasing the impact of this caching.

It is tempting to store the vocabulary in memory because doing so means that a disk access is avoided for every query term. However, if the vocabulary is large, keeping it in memory reduces the space available for caching of other information and may not be beneficial overall. Relying on the operating system swapping pages of vocabulary information is almost certainly a poor decision, as the distribution of hot query terms among pages is unlikely to be clustered in a useful way. An alternative is to manage the vocabulary in a B-tree as discussed earlier and to explicitly cache recently-accessed or frequently-accessed query terms in a separate small table.

Perhaps the most effective caching is of answer lists. If queries recur, it makes sense to store their answer lists rather than recompute them. Even keeping them on disk is effective as one short answer set can be fetched much more rapidly than can multiple inverted lists. For a typical search engine, most users will only view the first r answers, but it may be effective to keep the next r answers handy for the small percentage of cases in which they are requested.

Another form of caching is phrase indexing. Indexing all phrases is impractical, but using term-based inverted lists to identify which documents contain a phrase is expensive. The cost of phrase query processing could be significantly reduced if inverted lists are explicitly maintained for common term combinations. That is, the users themselves, via their querying behavior, can guide the evaluation as to which phrases might be worth explicitly storing when the index is next rebuilt.

One of the major bottlenecks of query evaluation is the need to fetch the documents that are presented in the answer lists; most search engines return not only a document identifier but a corresponding document summary or *snippet*. Typically these summaries are based on the query and thus cannot be computed ahead of time. Saving complete answer pages, including all of the required snippets, can thus be a dramatic saving.

Pre-Ordering. In the context of Web retrieval, techniques such as Google's PageRank can be used to determine a static score for each page to complement the dynamic score calculated by whatever similarity function is being used. The final rank ordering is then a blend of the static score which can be regarded as an a priori probability ordering that a page is relevant to queries in general, and a dynamic score which reflects the probability that a page is relevant to this particular query.

Static rankings are only useful in contexts where additional nontextual information can be used such as (in the case of PageRank) the link structure. The PageRank of a Web page is based on the link structure of the Web. Each page is assigned a score that simulates the actions of a random Web surfer, someone who either with probability p is equally likely to follow any of the links out of the page they are currently on, or with probability $1 - p$ jumps to a new page chosen at random. An iterative computation can be used to compute the long-term probability that such a user is visiting any particular page and that probability is then used to set the PageRank. The effect is that pages with many in-links tend to be assigned high PageRank values, especially if the pages that host those links themselves have a high PageRank; pages with low in-link counts, or in-links from only relatively improbable pages, are themselves deemed to be improbable as answers.

HITS is a similar technique which scores pages as both *authorities* and *hubs*, again using a matrix computation based on the connectivity graph established by the Web's hyperlinks. Loosely, a good hub is one that contains links to many good authorities, and a good authority is one that is linked to from many good hubs. Hubs are useful information aggregations and often provide some broad categorization of a whole topic, whereas authorities tend to provide detailed information about a narrower facet of a topic. Both may be useful to queriers.

Pages might also be assigned static scores contributions based on how similar they are to an amalgam of previous queries, or how often users of the search service have clicked through them to read their underlying content, or how deep they are in the directory structure of that particular site.

Where static weights are available from some source such as PageRank or HITS, they can potentially be used to eliminate some of the cost of ranking. For example, suppose that inverted lists are maintained in decreasing PageRank order. Then a top-r ranking could be rapidly computed by taking the Boolean intersection of the inverted lists corresponding to the query terms, terminating as soon as r documents containing all of the query terms have been found. These would be presented in decreasing PageRank order. If the user then requests more documents, presumably because the top r are unsatisfactory, then the system simply computes the top $2r$, and discards the first r of them.

User Feedback. Relevance feedback has been widely explored in the context of information retrieval research. In explicit feedback systems, users identify the answers that are of value (and perhaps others that are not), and this information is incorporated into a revised query, to improve the overall quality of the ranking. Much of this research assumes that queries are independent, and, in practical systems, it has proven difficult to gather relevance assessments from users.

However, in a system processing large numbers of queries, it is straightforward to identify which answers users choose to view. The action of a user clicking on a link can be interpreted as a vote for a document. Such voting can be used to alter the static weights of documents and thus their ordering in the ranking generated for subsequent queries even if the subsequent queries differ from the one that triggered the click though.

Index Terms. In structured documents, such as those stored as HTML, different emphasis can be placed on terms that appear in different parts of the document. For example, greater weight might be placed on terms that appear in a document's `<title>` tags or as part of an `<h1>` heading.

Another important difference between Web searching and more general document retrieval is that the anchor text associated with a link in one page is in many cases an accurate summary of what the target page is about. Links are usually manually

inserted, and unless they say "click here",[1] represent a succinct assessment to a human author as to the content of the target page. Indexing the anchor text as if it were part of the target page—perhaps even as if it were a title, or heading—can significantly improve retrieval effectiveness in Web search applications.

Another useful technique in Web searching is indexing the URL string itself. For example, the page at `www.qantas.com` shows the word Qantas as an image rather than a textual title and unless the URL were parsed into terms and indexed with the document, would risk not being searchable. Page importance can also be biased according to the length of the URL that accesses it on the assumption that URLs with fewer slashes lead to more important pages.

Finally, it is worth noting that the `<alt>` text associated with images is a useful indication of what the image content is, and indexing of `<alt>` text is part of the Google image search mechanism.

Manual Intervention. If some queries are particularly common, then a reasonable strategy for improving perceived performance is to manually alter the behavior of these queries. For example, it might make sense for the term `Quantas` to be manually associated with the Qantas page, given the high incidence of confusion over the exact name. By the same token, if the query `CNN` does not lead directly to `www.cnn.com` as a result of the ranking heuristic, then direct manipulation to make it so could well be appropriate. One of the most common queries in the search engine of one of our institutions is the single word `library`; sadly, the home page of the library is only the fourteenth highest ranked answer.

13. BIBLIOGRAPHY

The material presented in the previous sections is based on an extensive literature spanning more than 40 years. This section provides a high-level critical overview of that literature and can be regarded as further reading that augments the tutorial.

Our examination of past work is through the prism of current interests rather than a complete historical study. For example, in the first decades of work on information retrieval, both Boolean and ranked queries were widely investigated but, in the last twenty years, most work on search has focused on ranking; the research on Boolean querying is increasingly of historical interest only and is largely outside our scope. Note, too, that terminology has changed since the 1980s. In many older papers, *retrieval* was taken to mean Boolean matching. Scoring and ranking of matches was often described as *nearest neighbor searching*. Some retrieval papers focused on cluster retrieval, an activity that now receives relatively little attention.

Within the narrow area of inverted file indexing for text, this bibliography is reasonably complete. However, some minor papers have been omitted on grounds such as lack of availability (in particular, technical reports and theses); and, where a preliminary paper has later been expanded and republished, the preliminary paper is not cited. In the noncore areas, such as signature files, the bibliography is far from exhaustive. In these cases, we have focused on papers with innovations in index structures or query evaluation algorithms.

Some related topics are not considered. We do not explore vocabulary representations such as hash tables and B-trees or in-memory text search structures such as suffix arrays; and we do not discuss use of text indexes in other applications or special-purpose

[1] In late 2005, for the query `click here` Google returns the Adobe Acrobat Reader home page, the Macromedia Flash player download page, and the Apple QuickTime download page as the three top-ranked answers because of the hundreds of thousands of sites that point at the these pages using the anchor text "to obtain a copy, click here".