K

Kinetic Data Structures

1999; Basch, Guibas, Hershberger

Bettina Speckmann
Department of Mathematics and Computer Science,
Technical University of Eindhoven,
Eindhoven, The Netherlands

Problem Definition

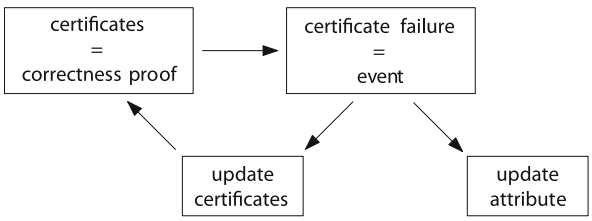
Many application areas of algorithms research involve objects in motion. Virtual reality, simulation, air-traffic control, and mobile communication systems are just some examples. Algorithms that deal with objects in motion traditionally discretize the time axis and compute or update their structures based on the position of the objects at every time step. If all objects move continuously then in general their configuration does not change significantly between time steps—the objects exhibit *spatial* and *temporal coherence*. Although *time-discretization* methods can exploit spatial and temporal coherence they have the disadvantage that it is nearly impossible to choose the perfect time step. If the distance between successive steps is too large, then important interactions might be missed, if it is too small, then unnecessary computations will slow down the simulation. Even if the time step is chosen just right, this is not always a satisfactory solution: some objects may have moved only slightly and in such a way that the overall data structure is not influenced.

One would like to use the temporal coherence to detect precisely those points in time when there is an actual change in the structure. The *kinetic data structure* (KDS) framework, introduced by Basch et al. in their seminal paper [2], does exactly that: by maintaining not only the structure itself, but also some additional information, they can determine when the structure will undergo a "real" (combinatorial) change.

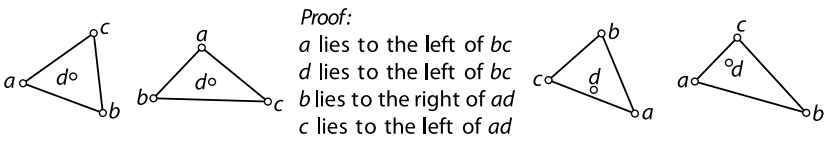
Key Results

A kinetic data structure is designed to maintain or monitor a discrete attribute of a set of moving objects, for example, the convex hull or the closest pair. The basic idea is, that although all objects move continuously, there are only certain discrete moments in time when the combinatorial structure of the attribute changes (in the earlier examples, the ordered set of convex-hull vertices or the pair that is closest, respectively). A KDS therefore contains a set of *certificates* that constitutes a proof of the property of interest. Certificates are generally simple inequalities that assert facts like "point c is on the left of the directed line through points a and b." These certificates are inserted in a priority queue (*event queue*) based on their time of expiration. The KDS then performs an event-driven simulation of the motion of the objects, updating the structure whenever an *event* happens, that is, when a certificate fails (see Fig. 1). It is part of the art of designing efficient kinetic data structures to find a small set of simple and easily updatable certificates that serve as a proof of the property one wishes to maintain.

A KDS assumes that each object has a known motion trajectory or *flight plan*, which may be subject to restrictions to make analysis tractable. Two common restrictions would be translation along paths parametrized by polynomials of fixed degree d, or translation and rotation described by algebraic curves. Furthermore, certificates are generally simple algebraic equations, which implies that



Kinetic Data Structures, Figure 1
The basic structure of an event based simulation with a KDS



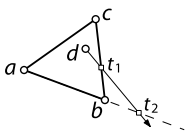
Kinetic Data Structures, Figure 2
Equivalent convex hull configurations (*left* and *right*), a proof that *a*, *b*, and *c* form the convex hull of *S* (*center*)

the failure time of a certificate can be computed as the next largest root of an algebraic expression. An important aspect of kinetic data structures is their on-line character: although the positions and motions (flight plans) of the objects are known at all times, they are not necessarily known far in advance. In particular, any object can change its flight plan at any time. A good KDS should be able to handle such changes in flight plans efficiently.

A detailed introduction to kinetic data structures can be found in Basch's Ph. D. thesis [1] or in the surveys by Guibas [3,4]. In the following the principles behind kinetic data structures are illustrated by an easy example.

Consider a KDS that maintains the convex hull of a set S of four points a, b, c, and d as depicted in Fig. 2. A set of four simple certificates is sufficient to certify that a, b, and c form indeed the convex hull of S (see Fig. 2 center). This implies, that the convex hull of S will not change under any motion of the points that does not lead to a violation of these certificates. To put it differently, if the points move along trajectories that move them between the configurations depicted in Fig. 2 without the point d ever appearing on the convex hull, then the KDS in principle does not have to process a single event.

Now consider a setting in which the points a, b, and c are stationary and the point d moves along a linear trajectory (Fig. 3 left). Here the KDS has exactly two events to process. At time t_1 the certificate "d is to the left of bc" fails as the point d appears on the convex hull. In this easy setting, only the failed certificate is replaced by "d is to the *right* of bc" with failure time "never", generally processing an event would lead to the scheduling and descheduling of several events from the event queue. Finally at time t_2 the certificates "b is to the right of ad" fails as the point b ceases to be on the convex hull and is replaced by "b is to the *left* of ad" with failure time "never."



Certificate	Failure time
a lies to the left of *bc*	never
d lies to the left of *bc*	t_1
b lies to the right of *ad*	t_2
c lies to the left of *ad*	never

Kinetic Data Structures, Figure 3
Certificate structure for points *a*, *b*, and *c* being stationary and point *d* moving along a straight line

Kinetic data structures and their accompanying maintenance algorithms can be evaluated and compared with respect to four desired characteristics.

Responsiveness. One of the most important performance measures for a KDS is the time needed to update the attribute and to repair the certificate set when a certificate fails. A KDS is called *responsive* if this update time is "small", that is, polylogarithmic.

Compactness. A KDS is called *compact* if the number of certificates is near-linear in the total number of objects. Note that this is not necessarily the same as the amount of storage the entire structure needs.

Locality. A KDS is called *local* if every object is involved in only a small number of certificates (again, "small" translates to polylogarithmic). This is important whenever an object changes its flight plane, because one has to recompute the failure times of all certificates this object is involved in, and update the event queue accordingly. Note that a local KDS is always compact, but that the reverse is not necessarily true.

Efficiency. A certificate failure does not automatically imply a change in the attribute that is being maintained, it can also be an *internal event*, that is, a change in some auxiliary structure that the KDS maintains. A KDS is called *efficient* if the worst-case number of events handled by the data structure for a given motion is small compared to the number of combinatorial changes of the attribute (*external events*) that must be handled for that motion.

Applications

The paper by Basch et al. [2] sparked a large amount of research activities and over the last years kinetic data structures have been used to solve various dynamic computational geometry problems. A number of papers deal foremost with the maintenance of discrete attributes for sets of moving points, like the closest pair, width and diameter, clusters, minimum spanning trees, or the constrained Delaunay triangulation. Motivated by ad hoc mobile networks, there have also been a number of papers that

show how to maintain the connected components in a set of moving regions in the plane. Major research efforts have also been seen in the study of kinetic binary space partitions (BSPs) and kinetic kd-trees for various objects. Finally, there are several papers that develop KDSs for collision detection in the plane and in three dimensions. A detailed discussion and an extensive list of references can be found in the survey by Guibas [4].

Cross References

- ► Fully Dynamic Minimum Spanning Trees
- ► Minimum Geometric Spanning Trees

Recommended Reading

1. Basch, J.: Kinetic Data Structures. Ph. D. thesis, Stanford University (1999)
2. Basch, J., Guibas, L., Hershberger, J.: Data structures for mobile data. J. Algorithms **31**, 1–28 (1999)
3. Guibas, L.: Kinetic data structures: A state of the art report. In: Proc. 3rd Workshop on Algorithmic Foundations of Robotics, pp. 191–209 (1998)
4. Guibas, L.: Modeling Motion. In: Goodman, J., O'Rourke, J.: (eds), Handbook of Discrete and Computational Geometry. CRC Press, 2nd ed. (2004)

Knapsack

1975; Ibarra, Kim

Hans Kellerer
Department of Computer Science, University of Graz, Graz, Austria

Keywords and Synonyms

Approximation algorithm; Fully polynomial time approximation scheme (FPTAS)

Problem Definition

For a given set of items $N = \{1, \ldots, n\}$ with nonnegative integer weights w_j and profits p_j, $j = 1, \ldots, n$, and a knapsack of capacity c, the *knapsack problem* (KP) is to select a subset of the items such that the total profit of the selected items is maximized and the corresponding total weight does not exceed the knapsack capacity c.

Alternatively, a knapsack problem can be formulated as a solution of the following linear integer programming formulation:

$$(KP) \qquad \text{maximize} \sum_{j=1}^{n} p_j x_j \tag{1}$$

$$\text{subject to} \sum_{j=1}^{n} w_j x_j \leq c\,, \tag{2}$$

$$x_j \in \{0, 1\}, \quad j = 1, \ldots, n\,. \tag{3}$$

The knapsack problem is the simplest non-trivial integer programming model having binary variables, only a single constraint and only positive coefficients. A large number of theoretical and practical papers has been published on this problem and its extensions. An extensive overview can be found in the books by Kellerer, Pferschy and Pisinger [2] or Martello and Toth [7].

Adding the integrality condition (3) to the simple linear program (1)-(2) already puts (KP) into the class of $\mathcal{NP}$-hard problems. Thus, (KP) admits no polynomial time algorithms unless $\mathcal{P} = \mathcal{NP}$ holds.

Therefore, this entry will focus on approximation algorithms for (KP). A common method to judge the quality of an approximation algorithm is its worst-case performance. For a given instance I define by $z^*(I)$ the optimal solution value of (KP) and by $z^H(I)$ the corresponding solution value of a heuristic H. For $\varepsilon \in [0, 1[$ a heuristic H is called a $(1-\varepsilon)$–*approximation algorithm* for (KP) if for any instance I

$$z^H(I) \geq (1-\varepsilon) z^*(I)$$

holds. Given a parameter ε, a heuristic H is called a *fully polynomial approximation scheme*, or an FTPAS, if H is a $(1-\varepsilon)$–approximation algorithm for (KP) for any $\varepsilon \in [0, 1[$, and its running time is polynomial both in the length of the encoded input n and $1/\varepsilon$. The first FTPAS for (KP) was suggested by Ibarra and Kim [1] in 1975. It was among the early FPTASes for discrete optimization problems. It will be described in detail in the following.

Key Results

(KP) can be solved in pseudopolynomial time by a simple dynamic programming algorithm. One possible variant is the so-called *dynamic programming by profits* (DP-Profits). The main idea of DP-Profits is to reach every possible total profit value with a subset of items of minimal total weight. Clearly, the highest total profit value, which can be reached by a subset of weight not greater than the capacity c, will be an optimal solution.

Let $y_j(q)$ denote the minimal weight of a subset of items from $\{1, \ldots, j\}$ with total profit equal to q. To bound the length of every array y_j an upper bound u on the optimal solution value has to be computed. An obvious possibility would be to use the upper bound $U_{\text{LP}} = \lfloor z^{LP} \rfloor$ from the solution z^{LP} of the LP-relaxation of (KP) and set

$U := U_{\mathrm{LP}}$. It can be shown that U_{LP} is at most twice as large as the optimal solution value z^*. Initializing $y_0(0) := 0$ and $y_0(q) := c + 1$ for $q = 1, \ldots, U$, all other values can be computed for $j = 1, \ldots, n$ and $q = 0, \ldots, U$ by using the recursion

$$y_j(q) := \begin{cases} y_{j-1}(q) & \text{if } q < p_j\,, \\ \min\{y_{j-1}(q),\ y_{j-1}(q - p_j) + w_j\} & \text{if } q \geq p_j\,. \end{cases}$$

The optimal solution value is given by $\max\{q \mid y_n(q) \leq c\}$ and the running time of DP-Profits is bounded by $O(nU)$.

Theorem 1 (Ibarra, Kim) *There is an FTPAS for (KP) which runs in $O(n \log n + n/\varepsilon^2)$ time.*

Proof The FTPAS is based on appropriate *scaling* of the profit values p_j and then running DP-Profits with the scaled profit values. Scaling means here that the given profit values p_j are replaced by new profits $\tilde{p}_j$ such that $\tilde{p}_j := \left\lfloor \frac{p_j}{K} \right\rfloor$ for an appropriate chosen constant K.

This scaling can be seen as a partitioning of the profit range into intervals of length K with starting points $0, K, 2K, \ldots$. Naturally, for every profit value p_j there is some integer value $i \geq 0$ such that p_j falls into the interval $[iK, (i+1)K[$. The scaling procedure generates for every p_j the value $\tilde{p}_j$ as the corresponding index i of the lower interval bound iK.

Running DP-Profits yields a solution set $\tilde{X}$ for the scaled items which will usually be different from the original optimal solution set X^*. Evaluating the original profits of item set $\tilde{X}$ yields the approximate solution value z^H. The difference between z^H and the optimal solution value can be bounded as follows.

$$z^H \geq \sum_{j \in \tilde{X}} K \left\lfloor \frac{p_j}{K} \right\rfloor \geq \sum_{j \in X^*} K \left\lfloor \frac{p_j}{K} \right\rfloor \geq \sum_{j \in X^*} K \left(\frac{p_j}{K} - 1 \right) = z^* - |X^*| K.$$

To get the desired performance guarantee of $1 - \varepsilon$ it is sufficient to have $\frac{z^* - z^H}{z^*} \leq \frac{|X^*|K}{z^*} \leq \varepsilon$.

To ensure this K has to be choosen such that

$$K \leq \frac{\varepsilon z^*}{|X^*|}\,. \tag{4}$$

Since $n \geq |X^*|$ and $U_{\mathrm{LP}}/2 \leq z^*$ choosing $K := \frac{\varepsilon U_{\mathrm{LP}}}{2n}$ satisfies condition (4) and thus guarantees the performance ratio of $1 - \varepsilon$. Substituting U in the $O(nU)$ bound for DP-Profits by U/K yields an overall running time of $O(n^2 \varepsilon)$.

A further improvement in the running time is obtained in the following way. Separate the items into *small* items (having profit $\leq \frac{\varepsilon}{2} U_{\mathrm{LP}}$) and *large* items (having profit $> \frac{\varepsilon}{2} U_{\mathrm{LP}}$). Then, perform DP-Profits for the scaled large items only. To each entry q of the obtained dynamic programming array with corresponding weight $y(q)$ the small items are added to a knapsack with residual capacity $c - y(q)$ in a greedy way. The small items shall be sorted in non-increasing order of their profit to weight ratio. Out of the resulting combined profit values, the highest one is selected. Since every optimal solution contains at most $2/\varepsilon$ large items, $|X^*|$ can be replaced in (4) by $2/\varepsilon$ which results in an overall running time $O(n \log n + n/\varepsilon^2)$. The memory requirement of the algorithm is $O(n + 1/\varepsilon^3)$. □

Two important approximation schemes with advanced treatment of items and algorithmic fine tuning were presented some years later. The classical paper by Lawler [5] gives a refined scaling resp. partitioning of the items and several other algorithmic improvements which results in a running time $O(n \log(1/\varepsilon) + 1/\varepsilon^4)$. A second paper by Magazine and Oguz [6] contains among other features a partitioning and recombination technique to reduce the space requirements of the dynamic programming procedure. The fastest algorithm is due to Kellerer and Pferschy [3,4] with running time $O(n \min\{\log n, \log(1/\varepsilon)\} + 1/\varepsilon^2 \log(1/\varepsilon) \cdot \min\{n, 1/\varepsilon \log(1/\varepsilon)\})$ and space requirement $O(n + 1/\varepsilon^2)$.

Applications

(KP) is one the classical problems in combinatorial optimization. Since (KP) has this simple structure and since there are efficient algorithms for solving it, many solution methods of more complex problems employ the knapsack problem (sometimes iteratively) as a subproblem.

A straightforward interpretation of (KP) is an investment problem. A wealthy individual or institutional investor has a certain amount of money c available which he wants to put into profitable business projects. As a basis for his decisions he compiles a long list of possible investments including for every investment the required amount w_j and the expected net return p_j over a fixed period. The aspect of risk is not explicitly taken into account here. Obviously, the combination of the binary decisions for every investment such that the overall return on investment is as large as possible can be formulated by (KP).

One may also view the (KP) as a "cutting" problem. Assume that a sawmill has to cut a log into shorter pieces. The pieces must however be cut into some predefined standard-lengths w_j, where each length has an associated selling price p_j. In order to maximize the profit of the log, the sawmill can formulate the problem as a (KP) where the length of the log defines the capacity c.

Among the wide range of "real world" applications shall be mentioned two-dimensional cutting problems, column generation, separation of cover inequalities, financial decision problems, asset-backed securitization, scheduling problems, knapsack cryptosystems and most recent combinatorial auctions. For a survey on applications of knapsack problems the reader is referred to [2].

Recommended Reading

1. Ibarra, O.H., Kim, C.E.: Fast approximation algorithms for the knapsack and sum of subset problem. J. ACM **22**, 463–468 (1975)
2. Kellerer, H., Pisinger, D., Pferschy U.: Knapsack Problems. Springer, Berlin (2004)
3. Kellerer, H., Pferschy, U.: A new fully polynomial time approximation scheme for the knapsack problem. J. Comb. Optim. **3**, 59–71 (1999)
4. Kellerer, H., Pferschy, U.: Improved dynamic programming in connection with an FPTAS for the knapsack problem. J. Comb. Optim. **8**, 5-11 (2004)
5. Lawler, E.L.: Fast approximation algorithms for knapsack problems. Math. Oper. Res. **4**, 339–356 (1979)
6. Magazine, M.J., Oguz, O.: A fully polynomial approximation algorithm for the 0–1 knapsack problem. Eur. J. Oper. Res. **8**, 270–273 (1981)
7. Martello, S., Toth, P. Knapsack Problems: Algorithms and Computer Implementations. Wiley, Chichester (1990)