CGI2015 manuscript No.
(will be inserted by the editor)

Efficient Multi-Constrained Optimization for Example-Based Synthesis

Stefan Hartmann · Elena Trunz · Björn Krüger · Reinhard Klein · Matthias B. Hullin

Abstract Digital media content comes in a wide variety of modalities and representations. Although they have obvious semantic and structural difference, many of them can be unwrapped into a one-dimensional parameter domain, e.g., time, one spatial dimension. Novel content can then be generated in this parameter domain by computing sequences of elements that are optimal according to an objective to be minimized and in addition satisfy a number of user-defined constraints. Examples for this type of content generation task are, audio synthesis, human motion synthesis or architectural texture synthesis. In that work we present a generalized algorithm for this type of content generation task. We demonstrate the potential of our technique on a selection of content creation tasks, namely the generation of extended animation sequences from motion capture libraries and the example-based synthesis of architectural geometry, such as buildings and street blocks.

Keywords example-based synthesis, data-driven animation, motion synthesis, building layouts

1 Introduction

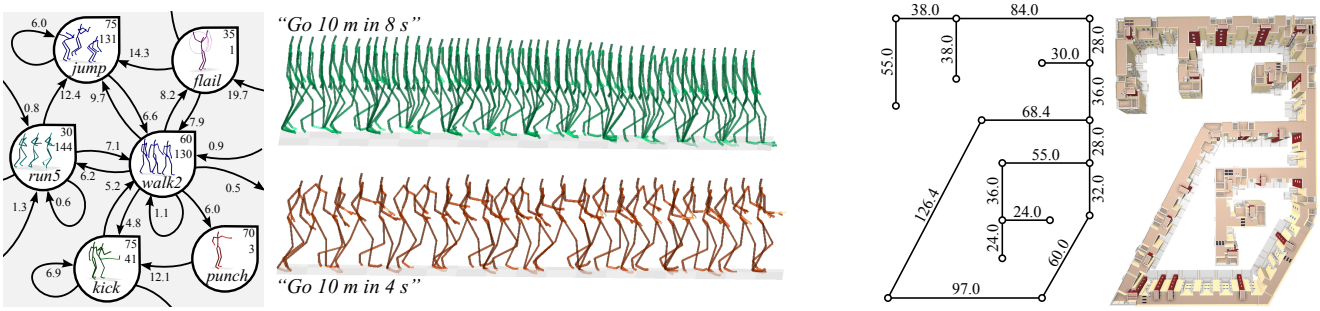
Intuitive and efficient design and editing of content for digital media in general, and computer graphics in particular, continues to be one of the most prolific subjects of investigation within our research community. Efforts to close the productivity gap of typical 2D and 3D modeling or animation workflows have given rise to novel models and user interfaces that make the editing of digital content more intuitive and efficient. With this work, we address an important sub-class of problems that occurs in virtually all fields of multimedia computing: the arrangement of atomic elements from a database into a sequence that is optimal under some problem-specific objectives and constraints. Whether it be animation sequences, architectural models or audio/video snippets, many types of media content have to be arranged along a one-dimensional parameter domain. In this work, these parameter domains include time or path length for motion synthesis applications, building length or height for architectural modeling. Typically the atomic elements carry multiple attributes, called resources, and relations (transitions) to other elements inside the database. They thus form a graph (Fig. 1(a)) from which novel content can be generated by computing one or multiple paths that minimize an objective function while satisfying one or more user-specified constraints. Fig. 1(b) illustrates this using a synthesized human locomotion sequence that consumes 10 meter of traveled distance under different time constraints.

In this paper, we propose to formulate the resource-constrained synthesis of new sequences from atomic elements as a general resource-constrained k shortest path (RCKSP) problem. While the case k=1 (RCSP) has already been addressed by the graphics community on several occasions, we uniquely focus on the more general task of finding multiple optimal solutions (a portfolio of solutions) and on satisfying multiple constraints. Both features are of great relevance to content generation problems in computer graphics. A unique property of our approach and key to its superior performance is that the graph traversal is guided by resource use, rather than the cost function as is usually the case.

The following are the main contributions of this work:

- We formulate the generation of sequences from a database of example elements as a resource-constrained k shortest path (RCKSP) problem.
- We introduce an additive resource-guided graph search technique for multi-constrained problems, where each constraint can be an interval or equality.

{hartmans,trunz,kruegerb,rk,hullin}@cs.uni-bonn.de
Institute of Computer Science II, University of Bonn, Friedrich-Ebert-Allee 144, 53113 Bonn, Germany



(a) (b) (c) (d)

Fig. 1 (a) Illustration of an augmented motion graph where nodes (motion segments) consume certain resources (number of frames, distance traveled in cm) and edge weights represent the transition cost. Our algorithm efficiently solves resource-constrained shortest path (RCSP) problems on such graphs, generating motion sequences that are optimal under a set of constraints. (b) Two sequences generated under different constraints (same distance but different target times). Depending on the time allowed, the result is a walking (top) or a running motion (bottom). (c) User-provided structure of a complex building with multiple nested T-junctions and a cycle. (d) Top view of a 3D model generated by the same algorithm from a database of building parts.

- We generalize our technique to more complex structures that may include cycles or T-junctions.
- We demonstrate the versatility of our approach using two established application scenarios: example-based modeling of architectural geometry, and the synthesis of sequences from a motion database.

2 Problem definition and prior art

In the following, we formulate the problem of computing optimal sequences subject to multiple constraints (equality and interval) with the focus on computing a set of k optimal solutions. We will proceed by reviewing the state of the art in solving the special case of $k = 1$, and conclude the section with an overview of prior work on applications in visual computing, also including literature hinting at further potential usage scenarios for our technique. Table 1 summarizes our literature review.

Optimal sequences as resource-constrained shortest paths. Assume a database D containing n elements. Allowing multiple occurrences of the same element, we compose a sequence X of elements from D. Every time element e is used, it consumes a certain amount of resources as given by its non-negative *resource usage vector* $\mathbf{r}(e) = (r_0, ..., r_m)(e)$, m being the number of different resources in our budget. The cost of placing e_i immediately before e_j in a sequence is given by the *transition cost function* $\delta(e_i, e_j) \geq 0$. The total resource usage, $\mathbf{R}$, and the total transition cost, Δ, for a sequence $X = (e_1, ..., e_p)$ of length p add up as

$$\mathbf{R}(X) = \sum_{i=1}^{p} \mathbf{r}(e_i) \quad \text{and} \quad \Delta(X) = \sum_{i=1}^{p-1} \delta(e_i, e_{i+1}). \qquad (1)$$

Let χ_c be the set of all *feasible* sequences that consume resources within certain given bounds,

$$\chi_c = \{X \in \chi \mid R_i^{\min} \leq R_i(X) \leq R_i^{\max} \; \forall \; i = 1, ..., m\},$$

where χ is the set of all sequences of elements of D. Our objective now is to find $\chi_{\text{opt}}^k = \{X_{opt}^1, \ldots, X_{opt}^k\}$, the set of k solutions, each of which is both feasible, i.e it satisfies the user defined resource constraints, *and* optimal with respect to the total transition cost after exclusion of known better solutions. In other words, the j^{th}-best solution X_{opt}^j can be defined recursively as follows:

$$X_{opt}^j = \underset{X \in \chi_c \setminus \{X_{opt}^1, \ldots, X_{opt}^{j-1}\}}{\operatorname{argmin}} \Delta(X) \qquad (2)$$

The constraint vectors $\mathbf{R}^{\min} = (R_1^{\min}, ..., R_m^{\min})$ and $\mathbf{R}^{\max}$ may be a mix of integers and real values, all non-negative. Their meaning depends on the application and can include the length of a generated sequence as well as other parameters (*"minimum number of balconies on city block"* or *"number of jumps in motion sequence"*). For now, we note the presence of both lower and upper bounds, which define an interval or an equality constraint if set to the same value (e.g., *"total duration of media playlist in seconds"*). Following the example of Lefebvre et al. [10], we interpret the above configuration as a graph with the database elements (and their resource usage vectors) at the states, and the transition cost $\delta(e_i, e_j)$ as the weight of the directed edge from state e_i to state e_j. The problem of minimizing the cost of a sequence as per Eq. 2 then immediately translates to computing the shortest path through the graph under the given resource constraints, a class of problem that has been investigated in operations research [16,3,20].

Discrete element sequences. In order to re-sequence audio content [22], or to synthesize curvilinear structured patterns by example [26], utilize dynamic programming to compute optimal sequences consisting of a fixed number of elements, which is either obtained explicitly by uniformly subdividing the input curve (Zhou) or implicitly by the length of the target song and the fixed size of elements retrieved from the source song (Wenner). Both algorithms are designed to

compute sequences where the number of output elements is known in advance. Lefebvre et al. [10] propose an algorithm to compose new images utilizing image strips of varying size. Their algorithm is able to compose a new image of a fixed size, without having to specify the number of image strips used in the final result in advance. Recently, Zhou et al. [25] proposed an algorithm for synthesizing structured vector patterns from an example pattern. The method uses dynamic programming to compute a topology aware layout followed by continuous optimization to compute an exact vector pattern ready for fabrication. Although our algorithm or problem formulation might be similar to existing solutions in graphics, we would like to emphasize, that we approach it in a very different way. First, we span the space of resource feasible solutions before optimization. This is possible, because we focus on additive resource usages. This turns out to be a significant advantage if the number of elements in the database is large. Second, our technique is capable of meeting an extended set of goals: (a) first, to satisfy multiple constraints which might be intervals; (b) second, to find k guaranteed optimal solutions; (c) third, generalized structures where elements can have more than two neighbors (t-junctions).

Method	Resource constraints / type	Key points / type	1.5D
Safonova[17]	None	**Yes/Class**	No
Lo[11]	None	**Yes/Class**	No
Lefebvre[10]	One/Equality	Yes/Element	No
Wenner[22]	One/Equality	**Yes/Class**	No
Zhou[26]	One/Equality	No	No
Ribeiro[16]	One/Interval	No	No
Ziegelmann[28]	Multiple/Upper	No	No
Zhu[27]	Multiple/Upper	No	No
Turner[20]	**Multiple/Equality**	No	No
Zhou [25]	**Two/Equality**	No	No
Ours	**Multiple/Interval**	**Yes/Class**	**Yes**

Table 1 Overview of existing methods that support different algorithmic features. The column headings are explained in Section 3 and the supplemental material.

Animation Sequence Synthesis. In computer animation, various shortest-path algorithms have been proposed to synthesize novel motions from motion capture databases. Kovar et al. [5] presented a greedy branch-and-bound approach to generate optimal walks through a motion graph following a pre-defined path. Their ideas were extended by Safonova and Hodgins [17], who introduced interpolated motion graphs and an efficient A^* search to generate smoother and more complex animation sequences along a path that may include environmental constraints similar to our key points described in Section 3. The exponential search space of this kind of problem was reduced by Lo and Zwicker [11]. By employing a bidirectional A^* algorithm and merging the two search trees by interpolation, they were able to synthesize human motions at interactive rates. However, we note that none of these approaches are designed to support resource constraints in order to handle specifications such as *"overcome a specific distance in 15 seconds, while jumping twice"*. Complementary to approaches based on graph search are compact generative probabilistic models to synthesize motions framewise [9] or utilizing morphable functions in combination with semantic annotations [13] from unsegmented prerecorded motion data.

3 Resource-constrained synthesis

3.1 Intermediate graph

In this section we describe and formulate a multi-constrained optimization approach for example-based sequence generation. The heart of our approach is an *intermediate graph* representation, which serves two major purposes. First, it transforms the original constrained problem into an unconstrained one. Second, it spans the search space, containing only feasible solutions, i.e. it only contains solutions that satisfy the constraints independent from the transition cost. To this end, we introduce discrete *resource consumption levels* (henceforth simply referred to as *level* l) that group subsequences of elements by their resource usage and impose a total order on the intermediate graph. After the intermediate graph has been built, the optimal or the k-optimal solutions can be computed using an existing k-shortest path algorithm such as the one by Eppstein [2].

Single-resource case. Let us first discuss the construction of the intermediate graph for problems that underlie a single constraint, leaving us with a scalar resource budget. The satisfaction of equality constraints is a major strength of our algorithm, so we will for now focus on this type of constraint. The intermediate graph is *directed* and *acyclic*, properties achieved by unrolling re-occurrences of elements in a sequence. Its nodes are represented as $N(e, l, r^c)$, where e denotes an element from the database, l represents the level on which the node will be placed, and r^c tracks the already allocated resource so far, including the element's own $r(e)$. Construction of the intermediate graph happens in three stages: *initialization*, *growing*, and *pruning*.

Initialization: Before the actual intermediate graph is built, first the number of levels L is determined from the single constraint R. In our setting, a constraint represents an application specific attribute A that needs to be satisfied by all feasible solution sequences in the set χ_c. The number of levels $L = R/GCD$ is determined by dividing R by the greatest common divisor (GCD) that the exemplars of D share for that specific attribute.[1] For the sake of simplicity, let us as-

[1] For real-valued constraints, if no GCD exists, we rescale/round the elements' resource usage values so that a GCD can be computed. In this way, the real-valued resource is transformed into one that consumes integer values and can be used in the synthesis setting. We perform this as a preprocessing step and store the result for each element $e \in D$.

sume $GCD = 1$ for the time being. Now, elements $e_i \in D$ that are allowed to be placed at the front and at the back of the solution sequences χ are inserted into the yet empty graph at levels $l_f = 0$ and $l_b = L - r^c(e_i)$. Their nodes are set to carry the information $N(e_i, l_f, r^c(e_i))$ for front elements and $N(e_i, l_b, L)$ for elements at the back, and they are connected by zero-cost edges from a source node s (front elements) or to a sink node t (back elements). The level can now be interpreted as a position between source and sink.

for all nodes $v_1 = (e, l_{st}, r^c(e))$ **do**
  for all possible successors e_{succ} of e **do**
    $l_{succ} \leftarrow l_{st} + r(e)$;
    $r^c(e_{succ}) \leftarrow r^c(e) + r(e_{succ})$;
    if $l_{succ} \leq l_{ts}$ **and** $r^c(e_{succ}) \leq L$ **then**
      $v_2 \leftarrow (e_{succ}, l_{succ}, r^c(e_{succ}))$;
      if node v_2 not exists **then**
        create node v_2;
      end if
      create edge $(v_1, v_2, \delta(e, e_{succ}))$;
    end if
  end for
end for

Function 1: Graph growing (forward step)

Graph growing: We grow the graph in a bi-directional fashion by alternating between *forward* expansion from the source s (listed in Function 1) and *backward* expansion from the sink node t, until a stopping criterion is met. In order to make the growing process easy to understand, the pseudo code presents the procedure in its simplest form. In the following, we explain its efficient implementation using two separate priority queues Q_f and Q_b, one for each expansion direction.

The fundamental difference of our algorithm to other graph search algorithms is that the nodes in Q_f and in Q_b are sorted by the current resource consumption instead of the path cost. More specifically, Q_f will prioritize nodes with the *least* resource consumption r^c, while Q_b processes with the *highest* consumption r^c first.

Without loss of generality, we start explaining the forward step. In order to expand the graph towards t, we extract the current top node $N_f(e_f, l_f, r^c(e_f))$ from Q_f, generate all possible successor nodes $N'(e_{succ}, l_{succ}, r^c(e_{succ}))$, and connect them with an edge of cost $\delta(e_f, e_{succ})$. The level l_{succ} of node N' is computed from the current top element of Q_f, namely $l_{succ} = l_f + r(e_f)$, and the overall resource consumption is increased by the successor's usage, $r^c(e_{succ}) = r^c(e_f) + r(e_{succ})$. The backward expansion works along the same lines, producing nodes that precede those currently stored in Q_b. Each time this step is executed, the current top node $N_b(e_b, l_b, r^c(e_b)$ of Q_b is extracted and predecessor nodes $N'(e_{pred}, l_{pred}, r^c(e_{pred}))$ are generated, adding edges with cost $\delta(e_{pred}, e_b)$. Here, e_{pred} is an element from the set of possible predecessors of e_b and the nodes level $l_{pred} = l_b - r(e_{pred})$ is computed by simply subtracting the resource usage $r(e_{pred})$ of the predecessor element e_{pred} from the current node level l_b. The resource consumption so far is updated accordingly, $r^c(e_{pred}) = r^c(e_b) - r(e_b)$. In case if a successor or predecessor node $N'(e, l, r^c(e))$ already exists only an edge labeled with the cost δ is inserted into the graph as shown in Function 1 is inserted. We label nodes by the direction in which they were created (forward, backward) and we restrict both steps to only expand nodes that were created from the same direction. The algorithm terminates when the level of the current top node l_f overlaps with the level l_b of the current top element in Q_b, or if both queues are empty. In the supplemental material, we prove that the worst-case complexity of the presented algorithm for a single constraint is $\mathcal{O}(Ldn)$, where n is the number of elements in the database and d is the maximum number of concatenation neighbors of an element.

Pruning. We do not produce nodes whose current resource consumption $r^c(e)$ would cause a resource *resource overflow/underflow*, because these nodes would not occur in any feasible solution at all. After the graph has been built, we apply a second pruning step recursively removing nodes inside the graph, which have no successor/predecessors. Such nodes result from not being connected to search graph of the opposite direction.

Multi-resource case. We now extend the single-resource solution to satisfy multiple equality constraints simultaneously. First, we revert from using the scalar resource usage r to the vector $\mathbf{r}$, ending up with a node definition of $N(e, l, \mathbf{r}_1^c(e))$. When incorporating multiple constraints the total order implicitly given by the single scalar resource consumption is lost. In order to re-establish a total order, the generated nodes are inserted lexicographically into the queues using their attached vectors tracking the currently consumed resources $\mathbf{r}^c(e))$. During the initialization, the number of levels is determined over the sum of all constraints ($L = \sum_{j=1}^{m} \mathbf{R}_j / GCD_j$), where GCD_j is the greatest common divisor that the exemplars of D share for the attribute j. The levels of the nodes are now computed by adding up the resources, arriving at $l_{succ} = l_f + \sum_{i=0}^{m} r_i(e_f)$ for the forward step. To keep track of consumption of each individual resource, the currently consumed resource vector $\mathbf{r}^c$ is updated as $\mathbf{r}^c(e_{succ}) = \mathbf{r}^c(e_f) + \mathbf{r}(e_{succ})$. Just as before, for the backward expansion step, we subtract the resources accordingly. Unlike in the single-resource case, multiple instances of an element may occur on a level, since different combinations of resources may result in the same sum. However, although multiple instances of a specific element might be placed on the same level, they can still be distinguished from the other instances, because of their unique

vector $\mathbf{r}^c(e)$ tracking the resources consumed until that element instance so far. Thus a new node is only generated if no node having the unique node signature $N(e, l, \mathbf{r}^c(e))$ already exists. An illustration of the step by step graph construction for a small concrete example is given in our supplemental material. If such a node $N(e, l, \mathbf{r}^c(e))$ already exists, only an edge is inserted. Further, we note that the relative scales of the resource components, and their order in the vector, have no influence on the final outcome of the algorithm. They do, however, affect the intermediate graph, the order in which the solutions are generated, and possibly the memory or runtime requirement of our algorithm.

Interval constraints. When interval constraints instead of equality constraints are given, only the second part of the initialization needs to be changed. Instead of one end level L, we now have an interval between L_{min} and L_{max}. Nodes for the elements allowed to be placed at the back of the solution sequences are then created for *all* levels from L_{min} to L_{max}. We observe that, despite the backward graph being expanded from a multitude of root nodes, in practice this bloats the graph less severely than expected. After all, during the expansion step, elements already present on a level can be re-used and will not need to be duplicated.

3.2 Structure: key points and 1.5D synthesis

A *sequence* in the sense of this paper is a succession of elements, which we associate with an arrangement along a primary dimension x, such as time or the length of a curve. Our algorithm treats this primary dimension as one out of several resources of which each element uses different amounts. Hence, if a certain time or length needs to be filled, the resulting sequence could consist of many short elements or just a few long elements. In order to give the user precise control over the synthesis, we use the concept of *structure*, which we define in a narrower sense than usual. The structure is the entirety of constraints on the synthesis.

It is specified by the user at and in between *key points* that live on the primary dimension of the content modality. For instance, the structure of an animation might contain the specification, "the actor should jump at time 5 seconds". It is not until the final sequence has been assembled that this key point will correspond to some i^{th} element that happens to end up at that particular time. The connection between structure and the sequence is therefore an implicit one. In this section, we will describe how our algorithm handles structural constraints in terms of key points, and how they can be used to assemble generalized ("1.5D") structures.

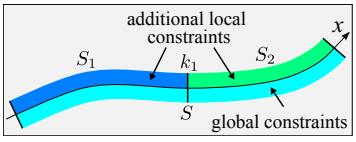


Fig. 2 Structure is a set of constraints defined on the primary dimension, x. Here, the key point k_1 divides the structure S into substructures $S_{1,2}$ that may each be constrained differently.

Key points divide an 1D structure into a set of *substructures* $S = \{S_1, \ldots, S_{m+1}\}$, where m is the number of key points (also see Fig. 2).

They may further enforce the localized occurrence of a specific element *class*, which is a harder task than just enforcing a particular element, because it means that adjacent substructures cannot be treated independently of each other. Additional local constraints may be specified for the domain covered by each of the substructures $S_i \in S$. For each S_i, the algorithm constructs a *subgraph* G_i, where elements placed at key points serve as front or back elements. All subgraphs G_i are now merged into the intermediate graph G, while the front elements of S_0 are connected by zero-cost edges to the source node s and the back elements of S_m connected by zero-cost edges to the sink node t.

1.5D Synthesis. Our framework described so far can also be used to synthesize structures in the shape of a binary tree or closed curves, while still guaranteeing a globally optimal result. To allow branching, we introduce a special type of key point with three neighboring elements. The major challenge here is to transform the tree into a 1D sequence and ensure the uniqueness of the elements that will end up being placed at key points. We solve this by performing an iterative longest path search starting from the root node of the initial tree (Figure 3(a)). We remove the path edges from the tree and store them as the *primary* structure. From the remaining connected components, we extract a set of *secondary* structures, and so on, until no more connected components are found. As a result we obtain a hierarchical set of structures that will be utilized to construct the intermediate graph. The actual graph construction starts with the primary structure that contains at least one key point, and subdivides it into multiple substructures, then constructs the initial graph for these as described above. For secondary and higher-order structures the subgraph construction differs slightly, because by definition these structures start with an element that has more than two neighbors. Setting up the subgraph of a higher order structure might cause element ambiguities, because the optimal element starting this structure might be a different one than chosen in the parent structure. To avoid such ambiguities we construct separate graphs for each element that is allowed to start this current structure. We recurse into child structures until we have a full set of subgraphs. In a final step, the subgraphs of child structures need to be merged into the graph of the parent structure. Here, we proceed as illustrated in Figure 3(b). Assume we have created separate graphs for substructure $k_1 \rightarrow S_4$,

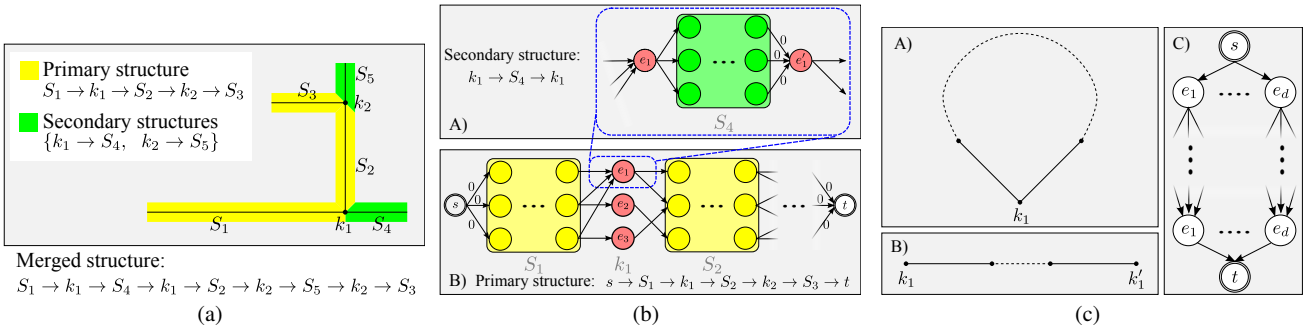
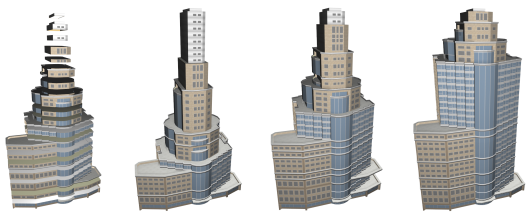


Fig. 3 (a) We decompose binary tree structures by recursively extracting longest paths from the structure graph. (b) Merging the subgraph of a child structure into its parent. (c) Closed curves are split by creating a virtual duplicate of one of the key points.

each starting with a different element e_i with more than two neighbors. In the parent structure, this corresponds to an element e_i^p, of which we create a *virtual duplicate* $e_i^{p\prime}$ that consumes zero resources and retains the outgoing edges from the original. On the original e_i^p, we replace the outgoing edges by new edges that connect to the front element of the substructure, whose back elements, in turn, are linked back to $e_i^{p\prime}$ via zero-cost edges. This procedure allows us to merge a child structure into the graph of the parent structure without causing element ambiguities. At the insertion point, the choice of a specific element (for instance, a T-shaped part) will thus depend on all its neighbors, rather than just on the usual two. In order to treat closed curves in a similar fashion, we simply split them at a key point (Figure 3(c)). The main difference to dead-end branches as described above is that here, there are real costs associated with the edges that close the cycle. In case if a cycle and t-junctions are present, we split and unroll the cycle at a key point and use the unrolled cycle as primary structure. Then the remaining higher order structures can be computed used the approach described above.

4 Application examples

In this section, we evaluate our algorithm on two distinct applications and present several results.



(a) Floor parts (b) FAR: 10.0 (c) FAR: 15.0 (d) FAR: 20.0

Fig. 4 Skyscrapers with a fixed height of 120 m and different floor-to-area ratios (FAR). Database: 18 elements, 80 transitions.

Architectural geometry. Using our technique, we synthesize urban blocks and individual buildings from a library of building parts. We show how using our concept of *structure*, global and local design goals can be implemented elegantly. One such example can be seen in Fig. 4, where we synthesized different skyscrapers from a set of floor parts. The resulting skyscrapers in Fig. 4 (b), (c) and (d) where forced to have the same height but the floor-to-area ratio (FAR, the total floor area divided by ground floor area) is varied. As one would expect choosing a small FAR results in a more slender building, while the larger the FAR is chosen, the more chubby the shape of the building becomes. Our databases are manually prepared by segmenting existing 3D buildings from Trimble Warehouse 3D. All parts are labeled as left and right *end parts*, *corners* with different angles, *middle/filling parts* and *T-shaped parts*, and annotated with different real-valued, integer attributes such as length, number of balconies, number of windows, number of doors etc. In addition, each part stores a polygon which represents the cross-section of the axis aligned cut for each side of the building part. The geometry of the building parts is scaled to quantize their length attribute to multiples of 0.2 m. The transition cost between two building parts is defined as

$$\delta(e_i, e_{i+1}) = |A_{i,r} - A_{j,l}|, \tag{3}$$

i.e., the area of cross-section disagreement between two adjacent parts, a measure of how well they fit together (see Eq. 3). The areas $A_{i,r}$ and $A_{j,l}$ represent the right cross-section of element e_i and the the left cross-section of element e_j, respectively. In all cases we arrange the neighboring costs in a precomputed lookup table and store it along with the database.

A second use case for architectural geometry is construction of more complex building layouts on the basis of a user defined footprint shape. We represent this footprint shape as a simple skeleton graph that describes the shape of the output building (see Fig. 1(c) (left)). Vertices in the skeleton represent key points, and edges connecting them

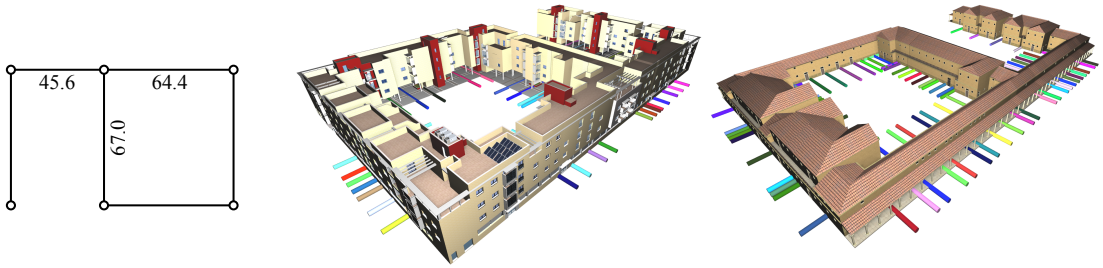


Fig. 5 Buildings obtained from the same structure using different styles. As indicated by the colored marks at part boundaries, the spatial composition of each sequence is highly dependent on the part database. This is why structure needs to be defined on the primary dimension rather than the sequence index (Section 3.2). Databases: 99 elements, 4950 transitions (left); 131 elements, 9170 transitions (right).

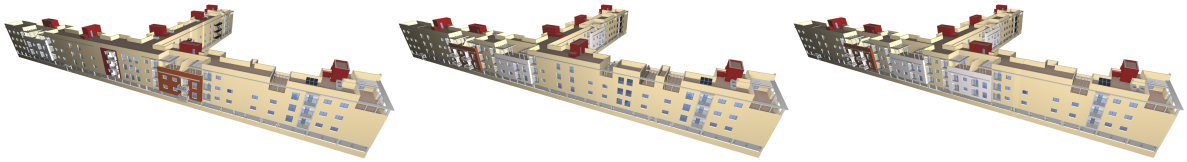


Fig. 6 Three of best solutions given the same input structure. Note that even the part at the T-joint can vary.

can be annotated with multiple additional constraints that are to be fulfilled in between, as well as an orthogonal direction vector to define the building front.

At each key point only a specific class of building parts is allowed to be positioned, which we determine by analyzing the number of outgoing edges of the skeleton vertices. If one edge leaves, either left or right end parts will be chosen, depending on the front direction of the edge. If two edges leave a vertex, *corner parts* depending on the angle between these edges are selected. *T-shaped* parts are selected from the database for vertices with three outgoing edges. From that input we compute a hierarchical structure according to Section 3.2 serving as input for the graph construction (see Section 3.1). From the resulting intermediate graph, the k-shortest-paths algorithm of Eppstein [2] computes either the optimal sequence X_{opt} or the k best sequences that contain valid arrangements of building parts according to the input skeleton and the globally and locally defined constraints. As the solution found by the shortest path algorithm is a sequential list of building parts, we need to arrange them according to the input skeleton as a final step. Each key point element in the solution is aware at which vertex of the input skeleton it needs to be positioned, and so we can compute partial buildings starting and ending with key point elements, and simply transform them to the corresponding skeleton edges.

Decoupling the database and the defined structure makes it easily possible to synthesize buildings of the same shape targeting a completely different style, as illustrated in Fig. 5. Note: Both buildings have exactly the same shape, but the number of elements chosen to realize the shape is completely different (colored cylinder represent the start of a new element in the resulting sequence). Variations can be achieved either by modifying the constraints as already demonstrated with the skyscraper model or by computing the k best solutions and selecting according to taste. Fig. 6 demonstrates the three best solutions according to the defined cross section disagreement cost function. All three buildings share exactly the same shape. By inspecting different limbs of the model and the parts selected at the t-junction and their surrounding, one we easily recognize that all three models vary their look locally. Such automatically generated variations may also serve as an inspiration to impose certain constraints on the next design iteration. For instance, the user may have noticed a balcony, but prefer to have it in a different location.

Length	# nodes constructed (runtime)
50 m	10K (0.51s)
80 m	13K (0.99s)
100 m	30K (1.5s)
200 m	66K (3.8s)
500 m	177K (9.8s)
1000 m	362K (22.8s)

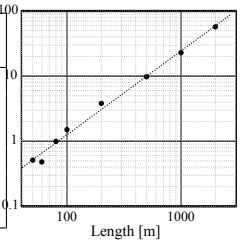


Fig. 7 Performance study: From a database of complete buildings (74 elements, 5402 transitions), we generated streets of varying length R and measured the total time required for building the intermediate graph and finding the shortest path. The image shows the solution for 80 m. We found the runtime to be roughly proportional to $R^{1.29}$.

Finally, we note that despite the simple nature of our user input (key points and constraints), this concept of structure allows for intuitive modeling even of very complex buildings (Fig. 1, 4, 5 and 6). All models shown took no more than a few seconds to generate on a standard desktop PC

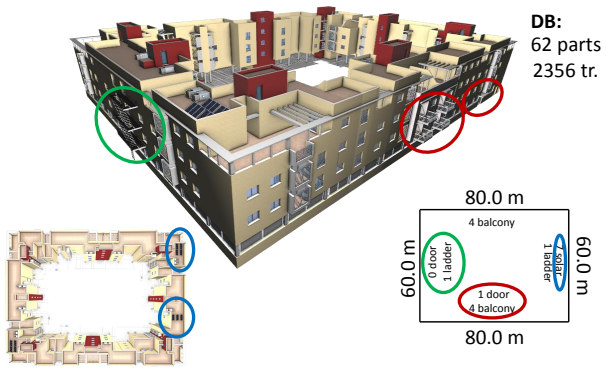


Fig. 8 A building synthesized with different local constraints.

(less than 3 s for the most complex example in Fig. 1), which makes our technique a candidate for interactive modeling sessions. In Fig. 7, we provide a rudimentary performance analysis across an extended range of problem sizes.

Discussion: From literature, we are aware that buildings and/or facades can be modeled using forward [15, 12] or inverse [1, 19] procedural modeling metaphors. Although they might be able to generate visually compelling results, we argue that these approaches are not directly suitable for modeling a combination of global and local constraints such as floor-to-area ratio or the specific occurrence of architectural elements as shown in our examples (see Fig. 8). Another possibility would be utilizing stochastic tiling as proposed by Yeh et al. [23], which might indeed generate plausible results; however, their method would struggle satisfying hard constraints.

Human motions. For this experiment, we used human motion segments extracted from the HDM05 Motion Capture Database [14]. However, the motions could be segmented using any method, for instance that by Zhou et al. [24], Vögele et al. [21] or Lan et al. [8]. The resulting database consists of 9709 segments with different motion classes such as *walk*, *jump*, *cartwheel*, etc. Each segment is annotated with the motion class it belongs to, and several properties such as start orientation, end orientation, trajectory length, animation length in frames. The transition cost between two motion segments $\delta(e_i, e_{i+1})$ is computed using the method presented by Krüger et al. [7] resulting in about 310.000 transitions and on average 32 possible successors for each motion segment. As input for the synthesis step, we define a path as spline curve, which the generated motion shall follow as closely as possible. Instead of approximating this rather special constraint using key points, we use it to illustrate that most components of our technique can be creatively bent without touching the core algorithm. A cost function might/needs to be defined to fit the application's need, however, it will typically not modify the core of our algorithm. For the motion synthesis example we add to penalize the distance to the path by extending the definition of the total cost Δ of a sequence to

$$\Delta = \sum_{i=1}^{p} \alpha\omega(e_i) + \sum_{i=1}^{p-1} (1-\alpha)\delta(e_i, e_{i+1}) \tag{4}$$

with ω being the distance between the sketched curve and the actor's root node projected to the ground plane. In addition, several key points and constraints might be defined to further structure the generated motion. Space and angle

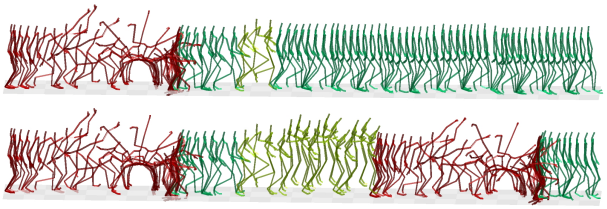


Fig. 9 Motions obtained for different semantic constraints on the same path. Database: 9709 elements, 310K transitions.

are discretized by 5 cm and 2.0 degrees, respectively. As the complete search graph is a product of the motion graph with all the possible (discretized) positions and orientations of the root of the animated character [17, 11] the complete setup of the graph would cost too much memory. We therefore construct the nodes located at a specific position in space (x, y, β) with β being the orientation of the character, on demand. In addition to the position and orientation of the root of the character, we store the current consumed k resources with each node represented by $N(e, x, y, \beta, r_i^c(e), \ldots, r_m^c(e))$ (see Section 1). After the graph setup is done we search for the k best solutions under the given constraints. The result is a sequence of human motion segments, which need to be concatenated in order to produce the final motion. This is achieved similar to Kovar et al. [5] using spline interpolation over a window of 5 frames to smooth the transition between motion segments. For further cosmetic improvement footskate cleaning [6] might be employed.

Discussion: Fig. 1 (b) show motion sequences synthesized with our algorithm. In both cases we forced the character to overcome a distance of 10 meters, however, in each sequence we utilized time as a second constraint in order to intuitively vary the speed of the character. Another result is shown in Fig. 9, where we specified walking along a straight line, but enforcing different constraints in each motion. More synthesized motion sequences are presented in our accompanying video. One advantage of our method is that we do not have to specify the order of the occurring actions that shall be performed along a path, enabling the designer to creatively explore the space of feasible character motions under the given constraints.

Performance Evaluation. In the supplemental material we sketch a simple algorithm, which we call *forward graph search* (FGS). Utilizing an implicit graph growing scheme, FGS generalizes and extends upon the algorithms presented by Lefebvre et al. [10] and Safonova et al. [17] to incorporate single and multiple constraints. We evaluate the performance of FGS against that of our technique using a subset of the HDM05 database that only contains motion segments from the *walk* class.

	Ours		
Length	Setup	Solve	FGS
5 m	5.5 s	0.16 s	19.1 s
6 m	18.8 s	1.0 s	67.3 s
7 m	75.2 s	3.7 s	160.7 s
8 m	164.6 s	11.9 s	452.7 s

Table 2 Performance evaluation (single constraint): Our algorithm vs. FGS

		Ours		
Length	Frames	Setup	Solve	Multiple Constraint FGS
5 m	146	3.4 s	0.17 s	84.7 s
6 m	146	23.1 s	0.38 s	185.6 s
7 m	187	108.1 s	4.7 s	834.9 s

Table 3 Performance evaluation (two constraints): Our algorithm vs. FGS extended to multiple constraints

Table 2 lists the runtimes for a single constraint. The goal is to synthesize a walking motion along a straight line, where the character is constrained to overcome different fixed distances. The results show that even for small problem instances our algorithm performs significantly better than the alternative FGS algorithm. In a second performance evaluation, we incorporate multiple constraints and compare the runtime of our approach against the multi-constraint FGS extension (see supplemental material). As the second constraint, we force the character to overcome the distance in a defined amount of time. The results listed in Table 3 show that although we construct the full graph, computing a solution to the problem takes significantly less time. We further evaluated the computation of k best solutions after we built the intermediate graph using the implementation of Eppstein [2]. We found computing multiple solutions given the intermediate graph only depends on the algorithmic complexity of Eppstein's algorithm. Since usually $k \ll n$, with n being the number of nodes, the runtime difference between computing 1 and 10^4 solutions is negligible.

5 Discussion and future work

Availability of annotated data: The outcome of any example based synthesis technique can only be as good as the data fed into it. In our case, the elements need to be meaningfully segmented and annotated with additional information, which is not included in the research databases we are aware of. Consequently, we had to manually prepare the 3D and motion data in order to make it usable for our purpose. The automatic and community-driven generation of example data bases are vibrant research topics and we are confident that in near future good example data will be available.

Constraining the search space: It is in the nature of hard constraints that they cut down the space of feasible solutions. This, of course, depends on many factors including the size of the database and the relative resource consumptions of the elements contained in it. In fact, in our experiments, we encountered cases where a very slight variation of equality constraints made all the difference between there being many solutions and none at all. Our algorithm performs best at equality constraints where suitable solutions exist. If there are none, for instance when trying to build a 50 m street out of a database of 20 m buildings, the failure can easily be detected by observing that the forward and backward trees do not connect. At that point, the user can be presented with several options to re-specify the problem, for instance by relaxing the constraint that caused the impossibility.

Characterization: Although we have presented a theoretical basis for handling more general structures including T-junctions under multiple constraints, our analysis of the algorithmic complexity is currently limited to the single constraint. From our experiments, we observe that the method scales well to rather complex structures including multiple T-joints. A rigorous analysis of the algorithmic and memory complexity in these scenarios is subject of future work.

Transfer to applications: The problem (RCSP) that our algorithm solves at its core is not specific to an application. However, due to the approximative way in which we handle resource usage, some creative experimentation may be needed in order to adapt the technique for a given content generation task. As more problems are solved using our technique, we therefore hope to gain the insight required to make recommendations on how to best deal with certain types of constraints. So far, we found that most are best formulated in terms of per-element resources, and others may map better to per-transition costs. Yet others may depend on the context and require a side tap into the algorithm, like the extra cost term we used to make the animation follow a path—an example we used to illustrate that the core of our technique is in principle flexible enough to allow for the injection of other kinds of constraints.

Outlook: So far, we have only started exploring the potential of our algorithm in real-world use cases, and there are many research directions that might be worth a closer look. For instance, we are not aware of any example-based synthesis techniques for multi-character animations (imagine a dancing couple), which could map well to motion graphs with T-joints as the actors split and re-join. Beyond the directions sketched in this paper, we plan to investigate the wider field of media computing. We see plenty of sequential prob-

lems that might benefit from our approach, from re-mixing of text, summarization of audio and video, to the generation of playlists or game level design [4, 18].

6 Conclusions

In this work, we presented a generalized approach for the synthesis of optimal sequences subject to multiple constraints. We showed how key points can be used to handle complex input descriptions (cycles, t-junction), while still ensuring global optimality. We demonstrated the versatility of our algorithm in two distinct application directions, and described how the respective settings can be cast to input for our algorithm. Our results illustrate the potential and in particular the flexibility of this approach. We believe that computer graphics research will benefit from studying RCSP/RCKSP-class algorithms more deeply, since many applications are based on data of some sequential nature, and demand for the satisfaction of multiple equality and interval constraints.

Acknowledgements

We thank AIF Projekt GmbH for their support through the AtEgoSim project, and Max Hermann for valuable discussions and his illustration of Fig. 2 and 3.

References

1. Martin Bokeloh, Michael Wand, and Hans-Peter Seidel. A connection between partial symmetry and inverse procedural modeling. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 29(4):104:1–104:10.
2. David Eppstein. Finding the k shortest paths. In *Proc. 35th Symp. Foundations of Computer Science*, pages 154–165. IEEE, November 1994.
3. Renan Garcia. *Resource constrained shortest paths and extensions.* PhD thesis, Georgia Institute of Technology, 2009.
4. Ian D Horswill and Leif Foged. Fast procedural level population with playability constraints. In *AIIDE*, 2012.
5. Lucas Kovar, Michael Gleicher, and Frédéric Pighin. Motion graphs. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 21(3):473–482, July 2002.
6. Lucas Kovar, Michael Gleicher, and John Schreiner. Footskate cleanup for motion capture editing. In *ACM SIGGRAPH Symposium on Computer Animation*, pages 97–104, 2002.
7. Björn Krüger, Jochen Tautges, Andreas Weber, and Arno Zinke. Fast local and global similarity searches in large motion capture databases. In *ACM SIGGRAPH Symposium on Computer Animation*, pages 1–10, July 2010.
8. Rongyi Lan and Huaijiang Sun. Automated human motion segmentation via motion regularities. *The Visual Computer*, 31(1):35–53, 2015.
9. Jehee Lee, Jinxiang Chai, Paul S. A. Reitsma, Jessica K. Hodgins, and Nancy S. Pollard. Interactive control of avatars animated with human motion data. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 31(5):491–500, 2002.
10. Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. By-example synthesis of architectural textures. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 29(4), 2010.
11. Wan-Yen Lo and Matthias Zwicker. Bidirectional search for interactive motion synthesis. *Computer Graphics Forum*, 29(2):563–573, 2010.
12. Paul Merrell and Dinesh Manocha. Constraint-based model synthesis. In *SIAM/ACM Conf. on Geometric and Physical Modeling*, pages 101–111, 2009.
13. Jianyuan Min and Jinxiang Chai. Motion graphs++: a compact generative model for semantic motion analysis and synthesis. *ACM Trans. Graph.*, 31(6):153:1–153:12, 2012.
14. M. Müller, T. Röder, M. Clausen, B. Eberhardt, Björn Krüger, and Andreas Weber. Documentation Mocap Database HDM05. Technical Report CG-2007-2, Universität Bonn, June 2007.
15. Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 25(3):614–623, 2006.
16. Celso C Ribeiro and Michel Minoux. A heuristic approach to hard constrained shortest path problems. *Discrete Applied Mathematics*, 10(2):125–137, 1985.
17. Alla Safonova and Jessica Hodgins. Construction and optimal search of interpolated motion graphs. *ACM Trans. Graph.*, 26(3), 2007.
18. Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. Rhythm-based level generation for 2d platformers. In *Conf. on Foundations of Digital Games*, pages 175–182, 2009.
19. Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. Metropolis procedural modeling. *ACM Trans. Graph.*, 30(2):11:1–11:14, 2011.
20. Lara Turner. Variants of shortest path problems. *Algorithmic Operations Research*, 6(2):91–104, 2011.
21. Anna Vögele, Björn Krüger, and Reinhard Klein. Efficient unsupervised temporal segmentation of human motion. In *ACM SCA*, July 2014.
22. Simon Wenner, Jean-Charles Bazin, Alexander Sorkine-Hornung, Changil Kim, and Markus Gross. Scalable music: Automatic music retargeting and synthesis. *Proc. Eurographics*, 32(2):345–354, May 2013.
23. Yi-ting Yeh, Katherine Breeden, Lingfeng Yang, Matthew Fisher, and Pat. Hanrahan. Synthesis of tiled patterns using factor graphs. *ACM Trans. Graph.*, 32(1):614–623, 2012.
24. Feng Zhou, F. De la Torre, and Jessica Hodgins. Aligned cluster analysis for temporal segmentation of human motion. In *IEEE Conf. on Automatic Face and Gestures Recognition*, 2008.
25. Shizhe Zhou, Changyun Jiang, and Sylvain Lefebvre. Topology-constrained synthesis of vector patterns. *ACM Trans. Graph.*, 33(6), 2014.
26. Shizhe Zhou, Anass Lasram, and Sylvain Lefebvre. By-example synthesis of curvilinear structured patterns. *Computer Graphics Forum*, 32(2):355–360, 2013.
27. Xiaoyan Zhu and Wilbert E. Wilhelm. A three-stage approach for the resource-constrained shortest path as a sub-problem in column generation. *Comput. Oper. Res.*, 39(2):164–178, February 2012.
28. Mark Ziegelmann. *Constrained shortest paths and related problems*. PhD thesis, Saarland University, 2004.