Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Diploma Thesis

Dynamic Data Structures for Scheduling

Bc. Jiří Kulovaný

Supervisor: Prof. Dr. Ing. Zdeněk Hanzálek

Study Programme: Open Informatics, Master

Field of Study: Software Engineering

May 5, 2013

Aknowledgements

I would like to thank to Ismael Ripoll, PhD., Polytechnic University of Valencia, for valuable consultations and support during my stay at the university and to Jakub Marecek, PhD. for problem discussions, data structures consultations and overall support provided.

Declaration

Abstract

This thesis presents an heuristic algorithm for dynamic scheduling of non-preemptive tasks with precedences on an arbitrary number of orthogonal resources, where the resource allocation is continuous. It is based on two data structures: critical-path task graph structure and constructive Container Loading Problem packing structure with an approach called “Skyline”. Both structures have been extended to support the algorithm. Algorithm complexity and computational results are shown.

Contents

Chapter 1

Introduction

1.1 Problem statement

In common schedulers in personal computers it is needed to schedule nonpreemptive tasks on one resource, which is usually the CPU. In large scale computation, scheduling tasks with precedences on more resources at one time can be useful, for example on processors, memory, communication buses, etc., while satisfying the maximum capacity constraint on each of these resources. The restriction on resource use can be extended further: the resource has to be allocated in a continuous block. Solving this for an arbitrary number of resources is computationally demanding.

This thesis describes a dynamic nonpreemptive task scheduler with precedences on an arbitrary number of resources with a continuous resource allocation constraint. The task scheduler allows modification of the resource allocation (*fitness*) function. There is an algorithm shown to dynamically schedule tasks with precedences on resources, while satisfying all the constraints. The program based on the algorithm is written in C++ while focusing on code portability. The program was tested to compile and run on Microsoft Windows and GNU/Linux platforms.

1.2 Motivation

The OpenCL is a framework for large scale parallel programming in heterogeneous systems. It defines abstract hardware on which the execution of tasks can be scheduled. Tasks executed in this system are called *kernels*. Each kernel has some resource demands and estimated execution time. In real life, the resource allocation plays quite a large role. For example NUMA (Non-Uniform Memory Access) technology introduces different memory access speed depending on the CPU/memory cell combination. This problem can be limited if tasks are scheduled with this in account, using resource allocation fitness function.

I hope this research might be useful in future versions of OpenCL framework or in any related scheduling fields.

1.3 Outline

This work is divided in the following chapters:

Task scheduler analysis chapter introduces the problem, two structures used for the problem solution are described and extensions to these structures are explained.

Proposed solution chapter shows the problem solution with the algorithm and illustrative example.

Performance overview chapter contains complexity analysis based on the underlying structures and also introduces basic framework for testing and shows actual run properties.

Conclusions chapter summarizes the achievement and shows possible future progression.

Chapter 2

Task scheduler analysis

The definition of the problem solved is as follows: Given d resources, each resource with maximum capacity r_i, determine the schedule of n non pre-emptive tasks, where each task $t_j = (p_j, R_j, P_j)$ has processing time p_j, set of resource constraints on each of the resources $R_j = \{r_1, ...r_d\}$ and set of precedence constraints on m other tasks $P_j = \{t_1, ..., t_m\}$. Two sets of tasks will be considered: the basic task set $T = \{t_1, ..., t_n\}$ and expanding set U of an arbitrary number o tasks $U = \{t_1, ..., t_o\}$. The precedence constraints in T can be only on tasks from set T, however the precedence constraints in U can be on tasks from both set T and U. The sets of tasks T and U must always form an acyclic directed graph, where the edges are precedence constraints in the form $\{a \rightarrow b\}$ denoting task a must be finished before the task b starts.

The goal is to schedule these tasks to minimize makespan and satisfy all the constraints, while each task has to allocate a continuous block of each of the resources.

The problem of dynamical scheduling tasks with precedences on an arbitrary number of resources while allocating continuous resource block is, as far as it is known to the author, new and uncharted. The paper *Scheduling with an Orthogonal Resource Constraint* [8] proposed an (2+ϵ)-approximation algorithm to an similar problem. The work described the approximation algorithm for scheduling tasks on one additional resource, e.g. memory cache and without precedence constraint.

Scheduling tasks on multiple orthogonal resources with continuous allocation constraint without precedence constraints reduces the problem to n-dimensional Strip Packing Problem (SPP). One of the ways to efficiently solve Strip Packing/Container Loading Problem is described by Allen et al. [1]. The basic algorithm represents the solution space by *skyline*. Details are consulted in section 2.2. Other approaches are reviewed by Bortfeldt et al. [2].

2.1 Task graph structure

Given the task graph of tasks to be scheduled a data structure is needed which can efficiently do the following:

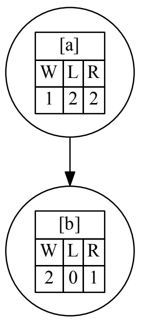
- Insert new subgraph to the existing graph.
- Answer "what is to be scheduled next" query.
- Manage the precedence time for scheduling.

The longest path structure proposed at [7] satisfies the first two conditions and is extended in this thesis to satisfy the third condition. The structure can answer the "what next" query given the task graph with precedences and claims the algorithm approaches optimality on the long run by the theorem below.

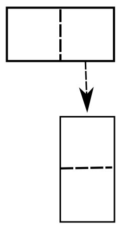
Theorem 1 (Papadimitriou and Tsitsiklis [9])**.** *Processing the job with the highest level ("largest sum of expected processing times along a path in the dependency graph") first, as soon as any machine is available, is asymptotically optimal with respect to weighted throughput, under certain conditions (*), among non-anticipative non-delay non-preemptive policies and non-anticipative non-delay preemptive policies with zero cost of preemption.*

The proof for any interested reader can be found in [7].

To show the extension of this structure to manage precedence times the basics of the algorithm will be sketched first.



(a) Task graph with emphasisis on properties, W is weight, L is level, R is resource demand



(b) Task graph with emphasisis on representation

Figure 2.1: Two equal task graph representation examples. $\{a \rightarrow b\}$ means task a has to finish before task b starts

2.1.1 Properties

As can be seen from the theorem 1, one needs to know *level* the for all the nodes, to be able to choose the best task to be scheduled. The definition of level can be expressed simplier than in that theorem - as a maximal length of the longest outgoing path from the node. Another property used in every node is *depth*, which represents the maximal length of ingoing path to the node. Last property of each of the nodes is *weight*, which represents the runtime of the task represented by the node.

All the nodes are kept in three directed acyclic graph structures, one primary and two auxiliary. Primary structure is called G and contains raw input nodes and edges and is the single point of truth after each operation. First auxiliary structure is called UL and is similar to G, except that all tasks in this structure have the same weight, i.e. 1. Second auxiliary structure is WL and contains nodes with assigned weight.

All the nodes with precedency constraint satisfied are stored in priority queue called *roots* with their *priority* in descending order, where $priority(t_i) = weight(t_i) + level(t_i)$.

2.1.2 Methods

New node insertion

The new node inserted is automatically root node and as such is inserted in the *roots* priority queue. It is also inserted in all three graph structures G, UL and WL.

New edge insertion

When new edge is inserted, the algorithm needs to update all nodes, which *level* or *depth* is changed by this new edge and, if necessary, delete the adjacent nodes from the *roots* structure. Update of the nodes in both UL and WL is done by methods `insertEdgeUpdateDownstream()` and `insertEdgeUpdateUpstream()`. Details for an interested reader on this methods can be found in [7].

Algorithm Schema 1: `insertEdge`(G, UL, WL, R, $u \to v$) [7]

```
1:  **Input:** Digraph $G$, weighted and unweighted auxiliary structures $UL$ and $WL$,
    priority queue $R$, edge $u \to v$ to be added
2:  **Effect:** Updated $G$, $UL$, $WL$, $R$
3:  $G = G \cup \{u \to v\}$
4:  **if** has(R, $v$) **then**
5:    remove($R$, $v$)
6:  **end if**
7:  $U = insertEdgeUpdateDownstream(UL, u, v)$
8:  $U = U \cup insertEdgeUpdateUpstream(UL, u, v)$
9:  $U = U \cup insertEdgeUpdateDownstream(WL, u, v)$
10: $U = U \cup insertEdgeUpdateUpstream(WL, u, v)$
11: **while** $U \neq \emptyset$ **do**
12:   $u$ =top($U$)
13:   update($R$, $u$)
14: **end while**
```

New sub graph insertion

To insert new sub graph U in the graph T, with set P^{ut} of cross-edges from U to T, the directed graph structures G, UL and WL of both graphs have to be merged. Then the sets of ready tasks are merged, except the tasks which represent vertexes in edges in P^{ut}. In the end, the precedence edges from set P^{ut} are inserted via new edge insertion method.

Best node query

For scheduling the best task, all that is needed to do is just to look at the first node at *roots*, since the roots are ordered by priority in descending order. Simple top query is done in $O(1)$.

Node removal

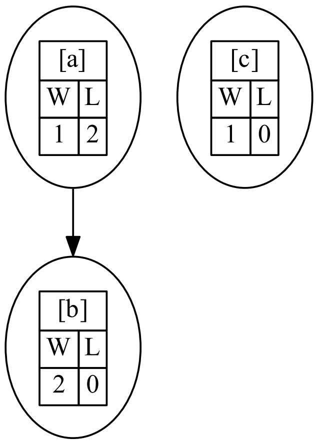
When scheduling a+ task, removal of the node representing the task from the structure is needed. This is done by removing the node from *roots* priority queue and by updating the graphs G, UL and WL. In the graphs we have to remove all outgoing edges from the node, recalculate all the *depth* and *level* properties of all dependent nodes by calling `popVertexUpdateDownstream()` method (details again in [7]). While removing edges, it is needed to check, if new roots should be inserted in the *roots* queue.

2.1.3 Extension of the structure

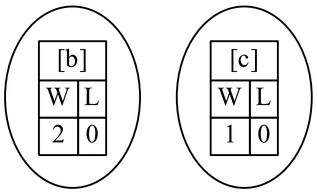
The basic algorithm is not ready for multiple co-running tasks. Best node query answer is the node with highest *priority*. This is correct as long as only one task can run at the same time. By allowing two or more tasks to run at the same time, we need to make sure that each task t_j is scheduled when all predecessors have finished.

Algorithm Schema 2: `popVertex`(G, UL, WL, R) [7]

```
1: Input: Digraph $G$, weighted and unweighted auxiliary structures $UL$ and $WL$,
   priority queue $R$
2: Effect: Updated $G$, $UL$, $WL$, $R$
3: $v = \text{top}(R)$
4: $U = \text{popVertexUpdateDownstream}(UL, v)$
5: $U = U \cup \text{popVertexUpdateDownstream}(WL, v)$
6: $S = \text{succ}(UL, v)$
7: while $S \neq \emptyset$ do
8:     $s = \text{top}(S)$
9:     $\text{push}(R, s)$
10: end while
11: while $U \neq \emptyset$ do
12:     $u = \text{top}(U)$
13:     $\text{update}(R, u)$
14: end while
15: $G = G \backslash \{u \to v\}$
```



(a) Initial structure $T = \{a, b, c, \{a \to b\}\}$



(b) Structure after one step $T' = \{b, c\}$

Figure 2.2: Task graph to show solution problem on more processors. Node properties described inside, where W is weight, L is level and $\{a \to b\}$ means a has to be finished before b is executed.

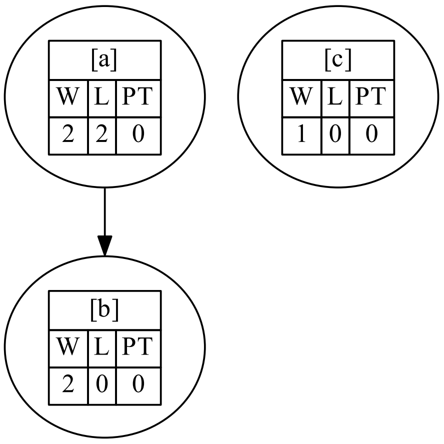
Problem example Lets have three tasks a and b and c with priorities $priority(a) = 3$, $priority(b) = 2$, $priority(c) = 1$ and precedency constraint $precedency(b) = a$ as can be seen in first part of figure 2.2. There are two processors in the system. The *roots* structure contains list of tasks in descending priority order $roots = \{a, c\}$. Best node query answers a, lets remove the node from structure. That gives us $roots = \{b, c\}$ as can be seen in second part of figure 2.2. Best node query answers b, which is incorrect for two processors. Correct task has to be ready at the same time, as was a.

Solution To solve this problem, all is needed to do is to add the execution time constraint. By extending the node properties with *schedulingTime* property, one can tell, when the node can be scheduled.

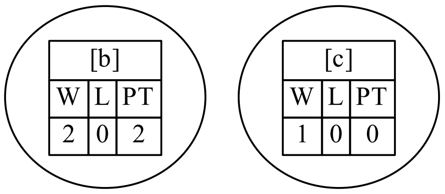
The node removal method will be extended to update all precedence dependant nodes with their new *schedulingTime*. Upon removing any node, it is needed to go through all its outgoing edges and set the *schedulingTime* to node schedule time + execution time.

The best node query will be updated to get the first available node, with *schedulingTime* greater than or equal to the query time. It is enough to go through the priority list and find the first node representing task, that can be scheduled.

Solution example Lets have three tasks a, b and c with priorities $priority(a) = 3$, $priority(b) = 2$, $priority(c) = 1$, precedency constraint $precedency(b) = a$ and newly added *schedulingTime* property equal to zero for all tasks, as can be seen in first part of figure 2.3. There are two processors in the system. The *roots* structure contains a list of tasks in descending priority order $roots = \{a, c\}$. Best node query for time zero answers a. While removing the a from structure, we update the *schedulingTime* of b to zero + $weight(a)$ (execution time of a). That gives us $roots = \{b, c\}$ and $schedulingTime(b) = weight(a)$, as can be seen in second part of figure 2.3. Second processor is still at time zero, thus the query for the best node is in time zero. The structure now correctly answers with c.



(a) Initial structure $T = \{a, b, c, \{a \to b\}\}$



(b) Structure after one step $T' = \{b, c\}$

Figure 2.3: Task graph to show problem on more processors. Node properties described inside, where W is weight, L is level, PT is precedency time and $\{a \to b\}$ means a has to be finished before b is executed.

2.2 Skyline structure

This section describes the Skyline structure proposed by Allen at al. [1] for Container Loading Problem solving. First the problem definition is shown, then the supporting structures used in Abstract Data Type (ADT) and the ADT itself. In the end of the section, the ADT implementations are presented.

2.2.1 Container Loading and Strip Packing Problems definitions

Container Loading Problem (CLP) and Strip Packing Problem (SPP) definition is as follows:

"A large d-dimensional (typically three dimensional) parallelepiped C is given, with dimensions $L_0^1, L_0^2, ..., L_0^d \in \mathbb{R}_+$ indicating, in the three-dimensional case its width, height, and length, respectively. A set B of n d-dimensional parallelepipeds to pack are also given with dimensions $L_i^1, L_i^2, ..., L_i^d \in \mathbb{R}_+$, again indicating width, height, and length in three dimensions; $X_i^1, X_i^2, ..., X_i^d \in \mathbb{R}_{\geq}0$, indicating the relative coordinates of their leftmost/lowest/deepest, etc., corner (i.e., that which is closest to origin); and $P_i \in \{0, 1\}$, indicating whether box i is packed or not. In all cases $i \in \{1, ..., n\}$. The objective is to position as many of the boxes B within the container C to maximise the volume utilisation, i.e.,

$$\text{Maximise} \left(\sum_{i=1}^{n} P_i \prod_{j=1}^{d} L_i^j \right). \tag{2.1}$$

The following constraints have to hold. The interiors of all the boxes should be disjoint (i.e., the non-overlapping constraint). This means that the boxes have to be non-overlapping in at least one dimension. The non-overlapping constraint can be expressed as

$$P_i \prod_{k=1}^{d} \text{overlap}(k, i, j) = 0 \quad \forall 1 \leq i < j \leq n, \tag{2.2}$$

where

$$\text{overlap}(k, i, j) = \begin{cases} 1 & \text{if } X_i^k < X_j^k + L_i^k \text{ and } X_j^k < X_i^k + L_j^k \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

The domain constraint states that the extremities of all placed boxes have to lie within the bounds of the container; i.e., the boxes cannot be positioned such that they are sticking out of the container, as formalised below:

$$P_i \sum_{k=1}^{d} \text{uncontained}(k, i,) = 0 \quad \forall 1 \leq i \leq n, \tag{2.4}$$

where

$$\text{uncontained}(k, i) = \begin{cases} 1 & \text{if } X_i^k < 0 \text{ or } X_j^k + L_i^k > L_0^k \\ 0 & \text{otherwise.} \end{cases} \tag{2.5}$$

The orthogonality constraint states that all placed boxes be positioned such that each side has to be parallel to a side of the container; i.e., if rotations of the boxes are permitted, then only those of 90° may be made. This constraint is implicitly held in the representation of boxes given here" [1]

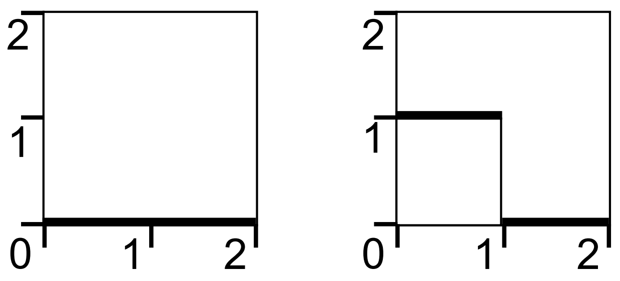


Figure 2.4: Skyline example.

Strip packing problem is similar problem to the CLP only with few differences. One of the dimensions (we can choose dth dimension without loss of generality) is unbounded or rather bounded by infinity and the objective function is to minimize the size used in this dimension.

2.2.2 The skyline approach

This approach is called skyline, because it resembles the real-life skyline. In real life, the skyline means the outline between earth and sky. Here it is the outline of the topmost packed d-dimensional hyper-rectangles. Their outline is the $d-1$ dimensional projection with assigned height (or dth dimensional proportion) property. The skyline is used as a representation of positions, where the box can be placed in the container. All space lower than the skyline is effectively disregarded in the packing process.

Demonstration is shown in two dimensions space with its properties width and height for first and second dimension respectively. Extension to more dimensions is as follows - second dimension will be dth dimension and first dimension represents the $d-1$ dimensions. In the example at figure 2.4, the skyline will be represented by thick line and the container and box borders by slim line. In the first part the empty container can be seen. The skyline thus covers the base of the container. The second part of figure 2.4 shows what happens after inserting a box with both height and width of size one. The skyline now consists of two parts, first part is on top of the inserted box, second is still at the base of the container.

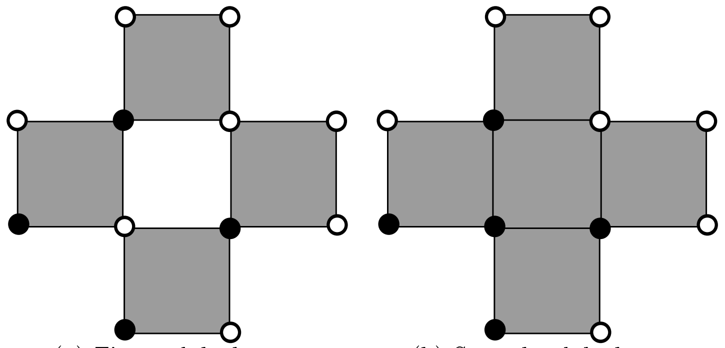

(a) First polyhedron. (b) Second polyhedron.

Figure 2.5: Orthogonal polyhedron example.

2.2.3 Rectilinear Polygon and simple gap

As the abstract data type interface described further on uses terms rectilinear polygon and simple gap, it is needed to introduce these constructs first. The rectilinear polygon is commonly known as a type of polygon, whose edges meet at right angle. Allen et al. [1] uses the rectilinear polygon representation with additional constraints called *simple gap*. Simple gap used further on will be convex parallelepiped, that can be described with a set of extreme coordinates by following formula: $rp = \{X_1^1, ...X_1^d, X_2^1, ...X_2^d\}$. X_1^i and X_2^j being first and second extreme coordinate in dimension i.

2.2.4 Orthogonal Polyhedron

Orthogonal polyhedron [1] is a finite union of full-dimensional hyper-rectangles. The whole skyline can be represented as n-dimensional orthogonal polyhedron or as a set of n-1 dimensional orthogonal polyhedrons for each height in skyline.

Definition: “Let $x = (x_1, ..., x_d)$ be a grid point. The elementary box associated with x is a closed subset of X of the form $\mathrm{B}(x) = [x_1, x_1 + 1] \times ... \times [x_d, x_d + 1]$. The point is called the leftmost corner of $\mathrm{B}(x)$. The set of boxes is denoted by B. An orthogonal polyhedron P is a union of elementary boxes, i.e. an element of 2^B.” [3]

[1] Orthogonal polytope generally for n-dimensional space.

2.2.4.1 Representation

To represent an polyhedron a set of vertexes with associated color function is used.

Color function "Let P be orthogonal polyhedron. The color function $\mathrm{c} : X \leftarrow \{0, 1\}$ is defined as follows: If x is a grid point, than $\mathrm{c}(x) = 1$ iff $\mathrm{B}(x) \subseteq P$; otherwise $\mathrm{c}(x) = \mathrm{c}(\lfloor x \rfloor)$." [3]

The grid point x is black if $\mathrm{c}(x)$ is 1 and white otherwise. Example can be seen on figure 2.5. The difference between first and second polyhedra is done by different color of one vertex. The first one has gap in the middle, because of the white color of the corresponding vertex. The second is full.

Bournez et al. [3] shows three different representation possibilities:

- Vertex representation
- Neighbourhood representation
- Extreme vertex representation

All representations are interchangeable. The difference between representations is mainly in the space used for storing each polyhedron, thus only vertex representation will be introduced here. This representation is based on keeping all vertexes, that form the polyhedra with their corresponding color. I.e. all the vertexes in example figure 2.7 are stored in a list.

2.2.4.2 Boolean operations

The boolean operation shown by Bournez et al. [3] on vertex polyhedra representation is straight forward. The approach will be shown on intersection (without loss of generality, the intersection can be replaced with any boolean operation). First, it is needed to introduce neighbourhood and vertex rules.

Neighbourhood The vertex x neighbourhood are verticles of a box lying between $x-1$ and x, that is verticles $N(x) = \{x_1 - 1, x_1\} \times ... \times \{x_d - 1, x_d\}$.

Vertex rules The vertex rules define, if the queried point is on an facet of the polyhedra, and if it is or it is not a vertex of the polyhedron. The distinction is made based on the color of neighbourhood. More details can be found at [3].

Let there be two polyhedra P_1 and P_2 with respective verticle sets V_1 and V_2. First, it is needed to generate all possible verticles created by these two polyhedrons. That means set $V = V_1 \cup V_2 \cup \{x : \exists y_1 \in V_1 \exists y_2 \in V_2 | x = max(y_1, y_2)\}$, where $max(x, y)$ is applied coordinatewise. Second, it is needed to determine the color of neighbourhoods of set V in both P_1 and P_2. Then the intersection function (or any other boolean function as stated above) is applied on all corresponding neighbourhood verticles. The result is checked by vertex rules and all points, that are verticles are stored in the new polyhedra P.

2.2.4.3 Rectilinear decomposition

The problem of rectilinear polyhedron decomposition to a set of simple gaps is as follows: Having an rectilinear polyhedron P, find the set of simple gaps G, where the union of the gaps covers P fully and exactly. The set G is maximal, gaps can overlap but no gap fully contains any other.

The idea in [1] to is generate all starting points, that is all possible points in the polyhedra (not only the vertexes, but also all points on grid defined by verticles). For each starting point s generate all simple gaps g_i contained by polyhedra starting from the largest ones and insert g_i to result set G, if the set doesn't already contain any gap, that fully contains the generated gap g_i.

Proposed extension: Not every point in the polyhedron is adept to be one of the starting points. We can remove all the white points as no gap can be generated from them and it is thus pointless to try generate any simple gaps from them. By removing all the white points, we lower the number of starting points by more than half.

2.2.5 Abstract Data Type

For scheduling, we need to effectively (complexity-wise) go through all the possible positions, where the box can be placed. The skyline at each single height can be viewed at as an $n-1$-dimensional rectilinear polyhedra. Since the boxes are n-dimensional parallelepipeds it is good to have all the positions pre-computed before trying to fit the boxes in the skyline. Based on this thought Allen et al. [1] proposed an ADT to the skyline approach, which works with unified representation of the skyline. Each height of the skyline is represented as a set of rectilinear polygons. As was stated in section 2.2.3, the simple gap representation of rectilinear polygon will be used.

ADT interface is defined as follows [1]:

- `getLowestGaps`() : {RectilinearPolygon}
- `getNeighbouringGaps`(rp : RectilinearPolygon) : {RectilinearPolygon}
- `splitGap`(rp : RectilinearPolygon, gap : RectilinearPolygon)
- `changeHeight`(rp : RectilinearPolygon, $newHeight$: Integer)
- `isContained`(rp : RectilinearPolygon, gap : RectilinearPolygon) : Boolean

`getLowestGaps()` function returns a list of all simple gaps that are located at the lowest position in the skyline (i.e., the deepest position in the container).

`neighbouringGaps()` function returns a list of the gaps in the skyline that are at higher positions than the gap being queried but touch its perimeter.

`splitGap()` function is used to split a *gap* into a number of new gaps. The number of new gaps ranges from 0 if the *box* fits the gap perfectly to any positive number if the *box* does not fill the gap entirely; e.g., its footprint is smaller than the gap.

`changeHeight()` function is used to change the height of part of the skyline to a new value, for example, when a box has been placed or an unusable gap is found.

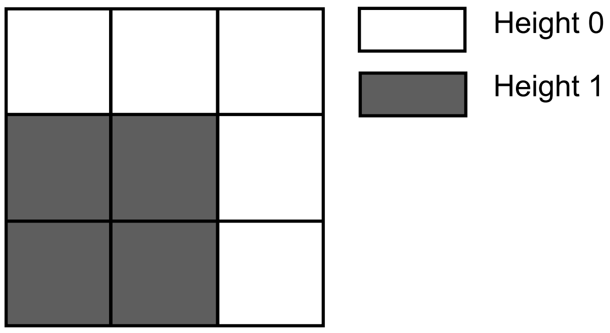


Figure 2.6: 2D Skyline example for 3D CLP.

`isContained()` function returns a Boolean value: true if the specified *box* fits within the specified *gap* and false otherwise.

2.2.6 ADT implementation overview

The skyline hidden behind ADT interface can be represented in many ways. Allen et al. [1] show five different ways, which then result in different algorithm complexities. The possibilities shown in the paper are using:

- Array representation
- Collision detection representation
- Plane representation
- Axis-aligned Bounding-box Tree representation
- Interval tree representation

Only the first and the last will be shown further on. The array representation is the easiest one, but with worst complexity. Interval tree on the other hand is the most effective one, with some great features (i.e. problem scaling does not affect complexity).

2.2.7 Array representation

Representing container space with an array is straight-forward. Lets create an $d-1$ dimensional array representing the resource space with size r_i for each resource. That is $A = \{a_1^1, ...a_1^{r_1}, a_2^1, ..., a_{d-1}^{r_{d-1}}\}$, where each cell a_i^j represents one position in skyline in dimension i and distance j. Each cell has assigned a number, which represents its height in the container. Example is shown in figure 2.6: two dimensional skyline representation in three dimensional container with assigned height property after inserting one two-by-two box with height one. Given this representation it is needed to meet the ADT interface.

`getLowestGaps()` To get all the gaps representing the lowest height in skyline, the whole array has to be traversed first, to get the height of the lowest gaps [2]. Next, the array has to be traversed again, to find all possible gap starting points and for each of the starting points the algorithm traverses the array space to get the largest possible gaps for the point.

`neighbouringGaps()` To find a list of neighbouring gaps it is only needed to inspect all array elements around the query gap.

`splitGap()` This operation does not need to be called in this representation. All the gaps are computed dynamically, thus splitting the gap does not affect the array at all.

`changeHeight()` Changing height of part of the skyline is simple. It is just a matter of updating all the array cells representing the query gap.

`isContained()` Because the size of each gap is determined on generation, only thing needed is to check the size of gap and query polygon in each dimension.

[2]This can be cached, but won't affect asymptotic complexity

2.2.8 Interval Tree representation

This approach represents the skyline by a set of simple gaps $d-1$-dimensional space. Each of the simple gaps is decomposed in a set of lines to improve complexity of some of the Skyline ADT queries.

2.2.8.1 Augmented Interval Tree

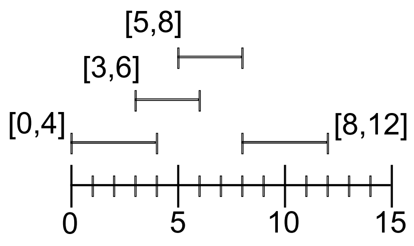
The interval i is defined as space between *low* to *high* endpoints in one dimension and, depending on the type of the interval, one or both extreme values. The types of the interval can be:

- Open $i = (low, high)$ None of the extreme values is part of the interval
- Half-open $i = [low, high)$ or $i = (low, high]$ One of the extreme values is part of the interval
- Closed $i = [low, high]$ Both the extreme values are part of the interval

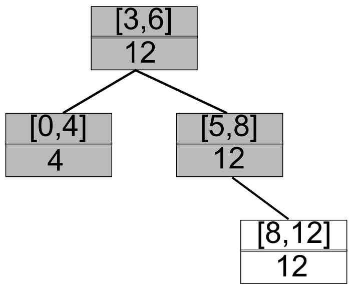
The interval tree structure can effectively answer the intersection query on a set of intervals with an interval or point in one dimension and can be extended to answer queries on lines in more dimensions. The naive approach to intersection query is to check all the lines with the query line and remember the ones, which intersected. The use of the interval tree reduces the query computational complexity from $O(n)$ to $O(\log n)$.

The methods supported are:

- `Insert()` which inserts interval to the structure.
- `Delete()` which deletes interval from the structure.
- `IntersectionQuery()` which returns list of intervals intersecting the query interval.



(a) Interval set



(b) Augmented red-black tree

Figure 2.7: Four interval set and associated augmented tree.

Cormen et al. [4] describes an effective implementation using augmenting red-black tree. The process of augmenting is extending the properties of any structure, to suit our needs. Each node n of the RB-tree will store one interval with both *low* and *high* properties. The key of the node n will be interval *low* endpoint. We store additional property *max* to the node n, which will be the maximum *high* property of subtree with node n as root. The `Insert()` and `Delete()` queries will update this property for all nodes affected.

An example of four intervals and associated augmented RB-tree can be seen on figure 2.7.

The `IntersectionQuery()` method shown in schema 3 on the interval tree root node n and query interval i will check, if the query interval is not to the right of all the nodes by comparing $i.low$ to the $n.max$ property. If it is to the right of all the intervals stored in the interval tree, the result is empty set. If it is to the left, the algorithm calls the `IntersectionQuery()` method on left subtree of node n and compares the interval stored in the node n to the query interval i. In next step queries the right subtree, if the query interval is not to the left from the starting point of interval stored in node n.

Higher dimensions The basic interval tree structure can be generalized to arbitrary number of dimensions assuming the lines are orthogonal. De Berg et al. [5] shows the generalization in second dimension for centered interval tree, which uses range tree

Algorithm Schema 3: `IntersectionQuery`(n, i)

```
1:  Input: Interval Tree root $n$, the query interval $i$
2:  Returns: Set of intervals intersecting $i$
3:  $Q \leftarrow \emptyset$
4:  if $n$ is not a leaf and $i.low \leq n.max$ then
5:      $Q \leftarrow$IntersectionQuery$(n.left, i)$
6:      if intersects$(n.interval, i)$ then
7:          $Q \leftarrow Q \cup n.interval$
8:      end if
9:      if $i.high \geq n.interval.low$ then
10:         $Q \leftarrow Q \cup$IntersectionQuery$(n.right, i)$
11:     end if
12: end if
13: return $Q$
```

with associated interval trees in every node, details consulted in depth by de Berg et al [5]. For augmented interval tree, the solution is to nest the interval trees. Having an interval tree in one dimension and nesting interval trees in every node n for next dimension for all the nodes in subtree where n is root.

2.2.8.2 ADT methods

As was stated in the beginning of this section, all the possible gaps are represented by rectilinear polygons d-dimensional space. Each of the rectilinear polygons can be decomposed in a set of lines. The interval tree is used to efficiently store this set of lines and then to do the intersection query. Another structure used is a height ordered list of the polygons from lowest to the highest to efficiently answer some of the queries.

`getLowestGaps()` To get all the gaps representing the lowest height in skyline it is needed to go through the ordered list of the polygons from the beginning and return all, that have the same height as the first polygon.

`neighbouringGaps()` To find a list of neighbouring gaps it is needed to query the interval tree with all the lines of the query polygon and from the result set of lines determine the set of gaps.

`splitGap()` To split a gap it is needed to:

- Remove from structure all gaps covering the query gap
- Unite them to one polyhedron by boolean masking operation described in section 2.2.4.2
- Substract the query gap from the polyhedron
- Decompose the polyhedron to simple gaps. Details have been consulted in section 2.2.4.3
- Insert the gaps back to the structure

`changeHeight()` To change height of part of the skyline it is needed to remove the query gap from the ordered list of polygons, update the new height of the gap and join with all the gaps at the new height. That means querying all neighbouring gaps on the same level, creating and decomposing polyhedron representing the gaps and reinsertion of all the new gaps.

`isContained()` All gaps are represented as rectilinear polygons, thus only thing needed is to check the size of gap and query polygon in each dimension.

Chapter 3

Proposed solution

In this chapter, the algorithm for scheduling is presented. It is based on the structures shown in the previous chapter. The scheduling algorithm combines the use of finding the best task to schedule presented in section 2.1 and scheduling it by placing in the skyline data structure (section 2.2). The tasks which are ready for scheduling are sorted by the task graph data structure in a descending priority order and their schedule time and resource allocation is determined by the skyline data structure position.

3.1 Detailed description

The full pseudo-code of the algorithm is presented in algorithm schema 4. The algorithm execution is following: The scheduler initializes properties at lines 4 to 8. The set G contains all simple gaps representing lowest height in skyline. The property $bestScore$ is set to zero, denoting no task has been found. The $bestTask$ property is set to empty task. $time$ is set to represent the lowest height in skyline, the height where the scheduling will take place.

When initialized, the algorithm finds the best task to be scheduled. Priority class, where at least one task can be scheduled needs to be found and in this class the best task to schedule is determined. That is done on lines 8 to 21 and all tasks can be queried. On lines 9 to 11 it is checked, if the best task has been found and (if it has been found), if the queried task T_i is in the same priority class as the task found. If the best score is not zero and the queried task is not from the same priority class, the best task to schedule has been found and this part ends. Otherwise, the queried task is checked on line 12, if it can be scheduled at the lowest height in skyline. If not, `continue` is called and the next task is queried.

At this moment, it is clear, that the task T_i can be scheduled. All gaps on lowest height are queried and checked, if the task T_i can be scheduled on the resources denoted by the gap G_j. If T_i can be scheduled on G_j, then the fitness function `evaluationScore()` is called. If the queried task T_i fits better in gap G_j than $bestTask$ in *bestGap*, the best combination so far has been found and it is assigned in $bestTask$, *bestGap* and $bestScore$.

When all relevant tasks have been queried, the algorithm checks if the best task has been found. This is done at line 26. If it has been found, the task is updated with position information on line 27. Then the modification of skyline is done, to represent the old skyline with the task inserted in it. On line 31, the task is removed from the taskgraph T and all dependant tasks are updated with *precedencyTime* property as stated in chapter 2.1.

If no task is suitable for scheduling at the time defined by lowest height of the skyline structure, then all lowest gaps are lifted to closest higher gaps and the algorithm is repeated.

Algorithm Schema 4: `schedule`(T, S)

```
1: **Input:** Taskgraph structure $T$, Skyline structure $S$
2: **Effect:** Empty $T$, Updated $S$
3: while not empty($T$) do
4:   $G \leftarrow$getLowestGaps($S$)
5:   $bestScore \leftarrow 0$
6:   $bestTask \leftarrow \emptyset$
7:   $time \leftarrow$ getLowestHeight($S$)
8:   for all $i$ such that $1 \leq i \leq |T|$ do
9:     if $bestScore$ != 0 and priority($T_i$) != priority($bestTask$) then
10:      break
11:    end if
12:    if $time <$ precedencyTime($bestTask$) then
13:      continue
14:    end if
15:    for all $j$ such that $1 \leq j \leq |G|$ do
16:      if isContained($T_i$, $G_j$) then
17:        $s \leftarrow$ evaluationScore($T_i$, $G_j$)
18:        if $s > bestScore$ then
19:          $bestScore \leftarrow s$
20:          $bestGap \leftarrow G_j$
21:          $bestTask \leftarrow T_i$
22:        end if
23:      end if
24:    end for
25:  end for
26:  if $bestScore \neq 0$ then
27:    position $bestTask$ in $bestGap$
28:    splitGap($bestGap$, $bestTask$)
29:    $height \leftarrow$ getHeight($bestGap$) + getHeight($bestTask$)
30:    changeHeight($bestGap$, height)
31:    updateSchedulableTime($T$, $bestTask$, $height$)
32:    popVertex($T$, $bestTask$)
33:  else
34:    for all $j$ such that $1 \leq j \leq |G|$ do
35:      $N \leftarrow neighbouringGaps(G_j)$
36:      changeHeight($G_j$, min(getHeight($N$)))
37:    end for
38:  end if
39: end while
40: return $S$
```

3.2 Illustrative example

The initial task graph to be scheduled can be seen in figure 3.1a. It consists of four tasks and three precedency constraints. The initial skyline can be seen in figure 3.1c, with skyline denoted by thick line. On x-axis, the resource r with capacity 3 can be seen. The y-axis represents the infinite time.

Each task scheduled will introduce different execution path of the algorithm. The overview of the execution path is explained first in high-level view, then the detailed view is shown.

For description convenience, few named lines or named parts of the algorithm will be introduced.

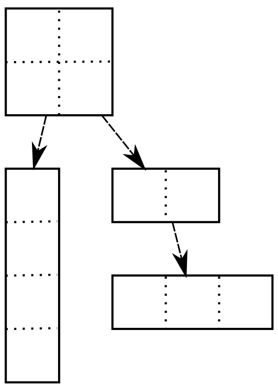
Main loop will denote the loop at line 3

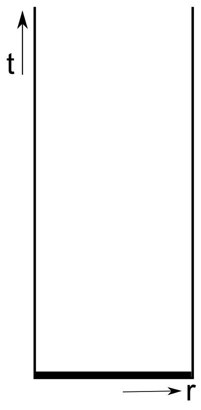
Best task search will denote the lines 8 to 25

Task placement will denote lines 26 to 31

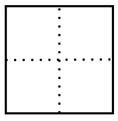
Lifting skyline will denote lines 33 to 37

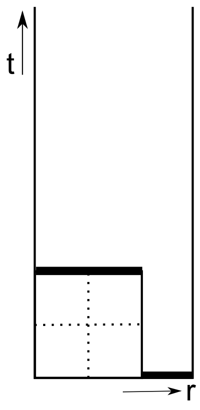
First task placement


(a) Initial task graph



(b) Initial skyline


(c) Task to be scheduled



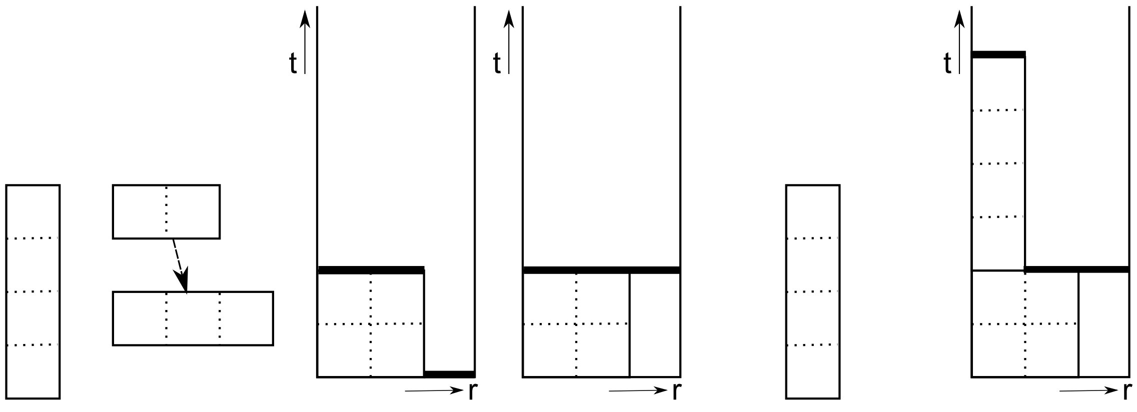
(d) Skyline after insertion

Figure 3.1: Algorithm first step.

This example shows simple insertion of task to skyline. The skyline split can be seen.

Given the task graph in figure 3.1a there is only possible task to be scheduled. The best task search thus gives the task shown at figure 3.1b. With the best task known, the task placement will be executed. The gap is split and height is raised to the height of the task, which is shown in figures 3.1c and d. The taskgraph is updated, the descendants of the node scheduled will have their *precedencyTime* set to 2. The skyline now consists of two gaps, first at height 0 and second at height 2.

Second task placement



(a) Initial task graph (b) Initial skyline (c) Skyline after one step (d) Task to be scheduled (e) Skyline after insertion

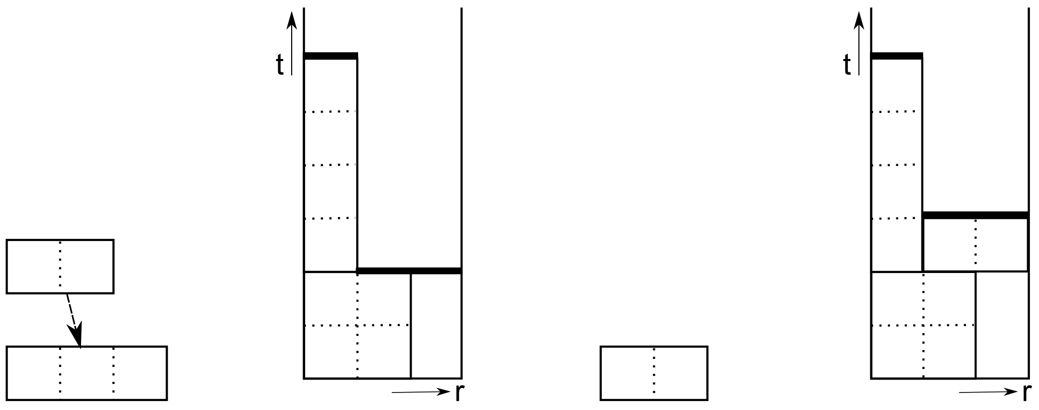
Figure 3.2: Algorithm second step.

This example shows what happens, if no task can be scheduled due to precedency time constraint. The lowest height of the skyline needs to be raised before the task can be scheduled.

The task graph now contains two root nodes, both with *precedencyTime* set to 2 and the lowest height of skyline is at 0. Since both nodes need to be scheduled at time higher, then the lowest height of skyline, the best task search will not find any task to schedule and task placement can't be executed. The lifting skyline is executed instead. The simple gap representing the skyline on the lowest height is raised to height 2, where it is joined with another simple gap. The skyline is again represented only by one simple gap, now at height 2 as can be seen in figure 3.2c.

The main loop is executed again and now the best task search will find the task shown at figure 3.2d and task placement will insert this task to the skyline, splitting and lifting the first part of the gap, as shown at figure 3.2e.

Third task placement

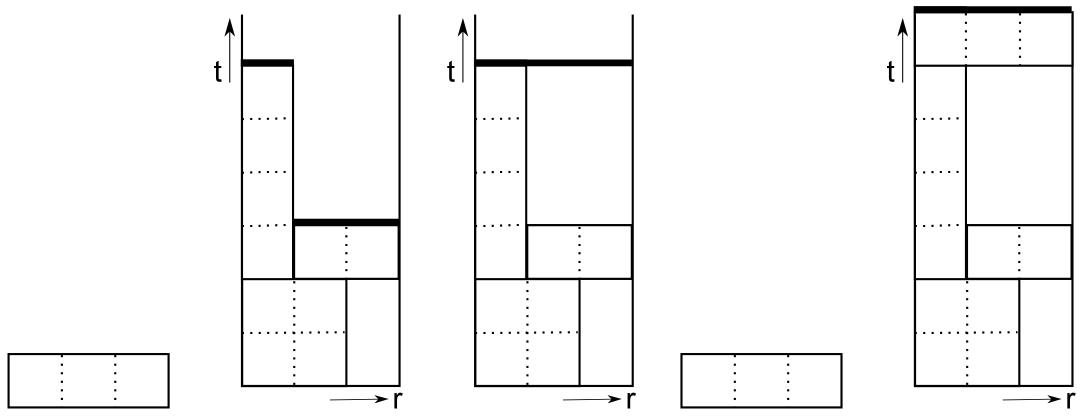


(a) Initial task graph (b) Initial skyline (c) Task to be scheduled (d) Skyline after insertion

Figure 3.3: Algorithm third step.

This example shows what happens if we have more heights of skyline available. Always only the lowest height is examined.

The task graph is now made of two tasks, where first task *precedencyTime* is 2. The lowest height of skyline is 2 and it is represented by one simple gap, representing resource of capacity 2. The best task search finds the task at figure 3.3b, which fits in the simple gap of resource capacity 2. The task is simply placed in the skyline and the simple gap representing the lowest height is raised atop the task as seen at figure 3.3d. The *precedencyTime* of the last task is set to 3.



(a) Initial task graph (b) Initial skyline (c) Skyline after one step (d) Task to be scheduled (e) Skyline after insertion

Figure 3.4: Algorithm fourth step.

Fourth task placement

This example shows what happens, if no task can be scheduled due to resource capacity constraint. The lowest height of the skyline needs to be raised before the task is scheduled.

The task graph is now represented only by one task with *precedencyTime* set to 3. The lowest height of skyline is at height 3, so the best task search will not fail on this, but it will fail on resource capacity. The simple gap on height 3 represents only capacity 2 and the task needs capacity 3. Thus lifting skyline will take place as can be seen at figure 3.4b and c. Then the main loop will be executed again and the best task search is successful as can be seen on figure 3.4d and the task is scheduled as can be seen at figure 3.4e.

Chapter 4

Performance overview

In this chapter the algorithm complexity is shown and then the runtime results are presented. The tests were executed on notebook HP ProBook 4530s with Intel Core i5 processor in Ubuntu Linux operating system.

4.1 Complexity analysis

First the methods from used structures are presented, then the methods invented for this algorithm.

External methods

Claim 1 ([1]). *The time complexity of* `getLowestGaps()` *is* $O(log(n) + k)$ *where* n *is the number of gaps and* k *is the number of gaps returned.*

The `getLowestGaps()` function returns a list of gaps that are located at the lowest position in the skyline. The gaps are queried in a depth-sorted list of gaps in $O(log(n) + k)$.

Claim 2 ([1]). *The time complexity of* `neighbouringGaps()` *is* $O(df * log(f) + k)$ *where* d *is the number of dimensions,* f *is the number of facets defining gap and* k *is the number of gaps returned.*

The set of neighbouring gaps is determined by query to the interval tree in $O(df * log(f) + k)$.

Claim 3 ([1]). *The time complexity of* `splitGap()` *is* $O(f^2 d^3 2^d)$ *where* f *is the number of facets defining gap and* d *is the number of dimensions.*

Splitting gap is case of removing the gap from both interval tree and ordered list taking $O(df \cdot \log(f))$ and $O(\log(n))$, then performing boolean masking operation taking $O(f^2 d^3 2^d)$ and insertion back to the tree and list again taking $O(df \cdot \log(f))$ and $O(\log(n))$.

Claim 4 ([1]). *The time complexity of* `changeHeight()` *is* $O(f^2 d^3 2^d)$ *where* f *is the number of facets defining gap and* d *is the number of dimensions.*

To change height of a gap, similar operations as in `splitGap()` are executed. The gap is removed from the structure, joined with all gaps at new height and reinserted in the ordered list.

Claim 5 ([1]). *The time complexity of* `isContained()` *is* $O(df)$ *where* d *is the number of dimensions and* f *is the number of facets defining gap.*

To check whether the box fits in the gap, one needs to check all $2d$ facets of the box to all f facets of the gap.

Claim 6 ([7]). *The time complexity of* `insertEdge()` *is* $O(|\delta|Q(|\delta|) + ||\delta||)$ *where* $Q(n)$ *is the complexity of insertion and extraction of an element in a priority queue with* n *elements,* $|\delta|$ *are vertices to be updated and* $||\delta||$ *are neighbouring vertices to be checked.*

The `insertEdge()` function inserts edge between two vertexes and updates the level property of all vertexes affected.

Claim 7 ([7]). *The time complexity of* `popVertex()` *is* $O(|\delta|Q(|\delta|) + ||\delta||)$ *where* $|\delta|$ *are vertices to be updated and* $||\delta||$ *are neighbouring vertices to be checked*

The `popVertex()` function deletes the vertex with no in-going edges and updates properties of all neighbours in the process.

Developed methods

`evaluationScore()` time complexity is $O(d)$, where d is number of dimensions.

Execution outline with complexities:

```
$O(d)$: In all dimensions
    $O(1)$: Query two extreme points of both gap and job
```

`updateSchedulableTime()` time complexity is $O(n)$.

Execution outline with complexities:

```
$O(n)$: For all descendants of the node
    $O(1)$: Update the schedulable time property
```

`getLevel()` time complexity is $O(1)$, time complexity of `getHeight()` is $O(1)$, time complexity of `getLowestHeigh()t` is $O(1)$

These functions returns properties of parameter objects and therefore have constant run time.

`schedule()` time complexity is $O(n^{2d+1}d^5 2^d)$

Follows the complexity calculation pseudo code with references to the algorithm on page 25. Each line is annotated with the cost and then the whole complexity for all branches is computed.

```
$O(ng)$: In the worst case all the gaps will be lifted for every
    box placed. The main loop on line 3 will be called with a
    bound of $ng$ times.
    $O(log(n)+k)$: getLowestGaps()
    $O(1)$: getLowestHeight()
    $O(n)$: The loop on the line 8
        $O(1)$: priority()
        $O(1)$: precedencyTime()
        $O(g)$: The loop on the line 15
            $O(d)$: isContained()
            $O(d)$: evaluationScore()
        The condition on line 26 will be true once for every
            task
            $O(d)$: To position box in the gap, we assign the
                position in each dimension
            $O(f^2d^3 2^d)$: splitGap()
            $O(f^2d^3 2^d)$: changeHeight()
            $O(n)$: updateSchedulableTime()
            $O(n\log(n))$: popVertex()
        The condition will be false for all other ocasions.
            $O(g)$: The loop on line 34
                $O(g)$: neighbouringGaps()
                $O(d^5 2^d)$: changeHeight()
```

Total complexity on the branches:

```
$O(ng\log(n) + n^2g^2d + nf^2d^32^d + n^2\log(n) + ng^2d^52^d + ng^3)$: The main loop
    on line 3.
    $O(ngd)$: The loop on the line 8
        $O(gd)$: The loop on the line 15
    $O(nf^2d^32^d + n^2\log(n))$ The condition on line 26 (final
        complexity)
    $O(gd^52^d + g^2)$: The condition will be false for all other
        ocasions.
        $O(gd^52^d + g^2)$: The loop on line 34
```

With the bound on the number of gaps being $O(g) = O(n^d)$ and facets being $O(f) \leq O(dn)$ this leads to: $O(n^{2d+1}d^52^d)$.

4.2 Computational results

This section shows the execution runtime for different kinds of graphs. First the algorithm makespan results are compared to the Standard Task Graphs for multiprocessor scheduling optimal makespan and then the time dependence on different properties of random graphs are shown.

All the values presented are the average over a set of repeated executions.

4.2.1 Standard Task Graph set for multiprocessor scheduling

Based on the standard task graph set for fair evaluation of multiprocessor scheduling algorithms by Tobita et al. [10], the makespan ratios are shown. For one resource type (processors) and if each of the tasks need exactly one processor, the problem solved by Tobita et al. [10] and the problem solved in this thesis are the same and the effectiveness can be compared.

In the tables 4.1, 4.2 the results on first 50 task graphs from each of the sets from the standard task graph set can be seen. Task graph with 50, 300, 500 and

750 nodes were tested. First the average, maximum and minimum makespan ratio is shown on the task graphs for 2, 4, 8 and 16 processors and then the overall results are summarized in table 4.3.

The makespan ratio for optimal makespan o given by Tobita et al. [10] and algorithm makespan a is computed as $ratio = \frac{100 \cdot o}{a}$ [%].

Nodes	50	50	50	50	300	300	300	300
Processors	2	4	8	16	2	4	8	16
Makespan ratio (avg) [%]	99,33	96,66	99,65	100,00	99,41	97,33	98,14	100,00
Makespan ratio (max) [%]	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
Makespan ratio (min) [%]	94,29	82,72	91,67	100,00	94,76	89,11	91,59	100,00

Table 4.1: Results on standard task graphs

Nodes	500	500	500	500	750	750	750	750
Processors	2	4	8	16	2	4	8	16
Makespan ratio (avg) [%]	99,52	97,37	98,26	99,98	99,49	97,32	97,84	100,00
Makespan ratio (max) [%]	100,00	100,00	100,00	100,00	100,00	100,00	100,00	100,00
Makespan ratio (min) [%]	97,23	90,46	88,75	99,04	97,53	89,18	88,66	99,78

Table 4.2: Results on standard task graphs

Makespan ratio (avg) [%]	98,81
Makespan ratio (max) [%]	100,00
Makespan ratio (min) [%]	82,71

Table 4.3: Overall results

4.2.2 Graphs with constant node to edge ratio

The graphs presented here are randomly generated graphs based on the $G(p, N)$ model proposed by Erdös et al. [6]. The graphs used for comparison are generated with these properties: Task dimension d, task amount n, node to edge ratio ne and size ratio s.

The node ratio is computed in following way: $ne = \frac{n}{|E|}$. The size ratio is used to scale the whole problem, that means actual edge size es is calculated for some constant c as $es = s \cdot c$.

The effect of the number of nodes on execution time

The graph presented on figure 4.1 shows function of time t on amount of nodes, such as $t = \mathrm{f}(n)$. The size ratio is constant and equals to $s = 1$. The dimension and edge ratio combinations used are as follows: $\{d = 3, ne = 4\}$, $\{d = 3, ne = 8\}$, $\{d = 2, ne = 4\}$.

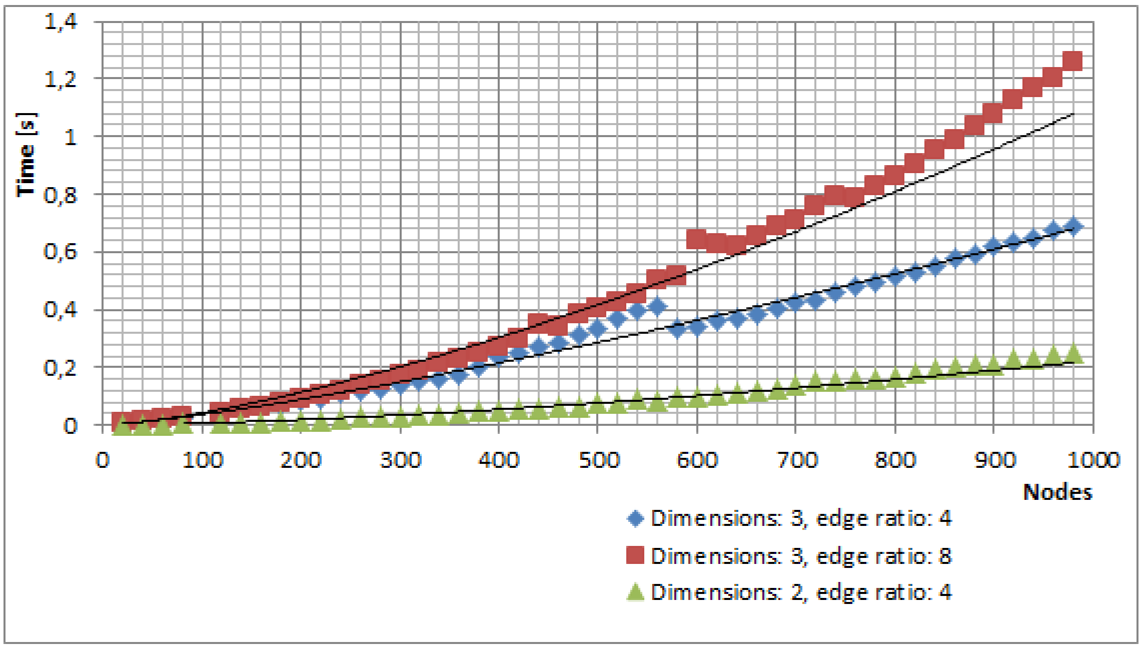


Figure 4.1: $t = \mathrm{f}(n)$ for $\{d = 3, ne = 4\}$, $\{d = 3, ne = 8\}$, $\{d = 2, ne = 4\}$

The effect of the length scale ratio on execution time

The graph presented on figure 4.2 shows function of time t on the size ratio, such as $t = \mathrm{f}(s)$. The dimension used is $d = 2$, amount of task queried $n = 100$ and edges to nodes ratio $ne = 4$.

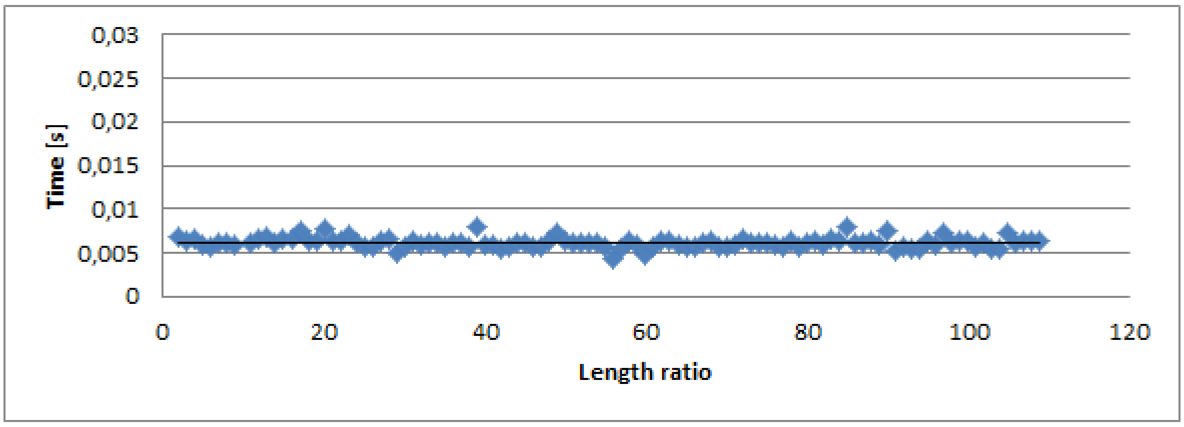


Figure 4.2: $t = \mathrm{f}(s)$ for $d = 2$, $n = 100$, $ne = 4$

The effect of the edge ratio on execution time

The graph presented on figure 4.2 shows function of time t on the length scale ratio, such as $t = \mathrm{f}(ne)$. The dimension used is $d = 3$, amount of task queried $n = 100$ and size ratio $s = 1$.

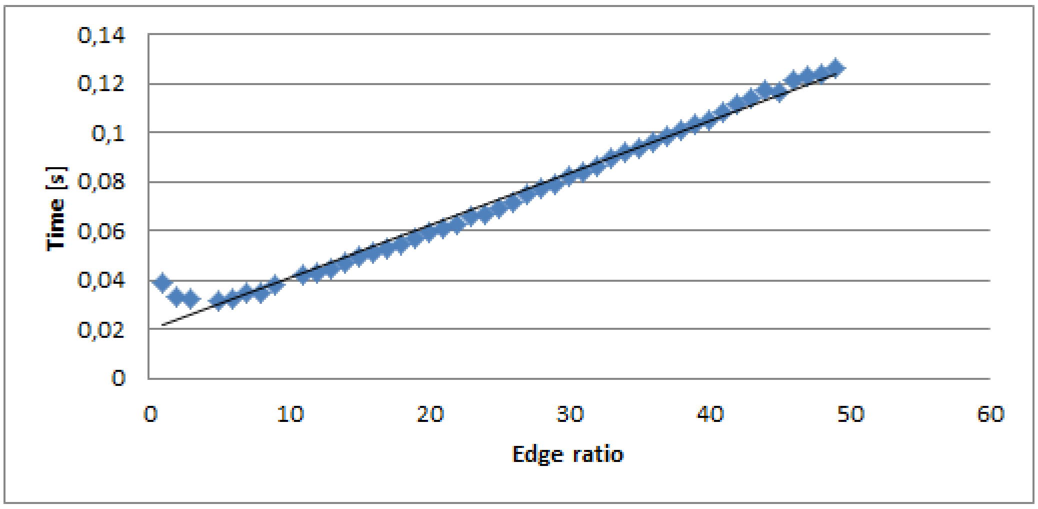


Figure 4.3: $t = \mathrm{f}(ne)$ for $d = 3$, $n = 100$, $s = 1$

4.2.3 Result discussion

As far as the author knows, there is no such algorithm on the bibliography, thus general comparison to previous works is not possible. The results can be compared if the problem is limited on two resources (time and processors) and the tasks allocate exactly one processor, as was stated in the beginning of the chapter. The algorithm makespan to optimum makespan is compared. The worst makespan ratio is 82,71 % to the optimum and average case is 98,81% on the set of standard task graphs shows the advantage of using the critical-path structure proposed by Marecek et. al [7].

The execution time function is shown for graphs with different properties. First, the time function on the amount of nodes is shown on figure 4.1 and the polynomial time complexity can be seen. Then the time function on problem scale is shown. Thanks to Interval tree based Skyline structure, the execution time function is constant for different problem sizes. Last graph shows the linear progression of time function on the number of edges.

Chapter 5

Conclusions

The original goal was to develop an algorithm for dynamic non-preemptive task scheduler with precedencies on an arbitrary number of resources with a continuous resource allocation constraint and the goal was achieved. The presented algorithm provides a solution to this problem, relying on two different data structures - the critical path task graph structure proposed by Marecek et al. [7] and Skyline structure proposed by Allen et al. [1]. In order to support the algorithm, some extensions to the structures were needed and are presented in the second chapter.

The problem solved belongs to the NP-hard class. The algorithm shown provides an approximate solution, with complexity bound being polynomial to the number of the nodes, when the resource count is considered constant. The relation between time and problem size scale is constant and between time and amount of edges is linear.

The result makespan for task graphs tested is quite flattering. The tests have shown that the ratio of worst makespan to optimal makespan is at 82% to optimum and the average case is above 98,81% on the data sets provided for fair multiprocessor scheduling.

As far as I know, the problem solved is not on the bibliography and thus this is the first algorithm to do so.

5.1 Future research

During the development I discovered several research lines that can be explored. To mention a few:

- Extend the structure to allow for richer task graph models, i.e with tardiness, multiple task alternatives, ...
- Extend allocation functions:

  Some resource combinations might be prioritized

  The function might modify the execution time based on allocation or prioritize with this taken in account

Bibliography

[1] ALLEN, S. D. – BURKE, E. K. Data Structures for Higher-Dimensional Rectilinear Packing. *INFORMS Journal on Computing*. Summer 2012, 24, 3, s. 457–470. doi: 10.1287/ijoc.1110.0464. URL: <http://joc.journal.informs.org/content/24/3/457.abstract>.

[2] BORTFELDT, A. – WäSCHER, G. Container Loading Problems - A State-of-the-Art Review. FEMM Working Papers 120007, Otto-von-Guericke University Magdeburg, Faculty of Economics and Management, April 2012. URL: <http://ideas.repec.org/p/mag/wpaper/120007.html>.

[3] BOURNEZ, O. – MALER, O. – PNUELI, A. Orthogonal Polyhedra: Representation and Computation, 1999.

[4] CORMEN, T. H. et al. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.

[5] BERG, M. et al. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000. URL: <http://www.cs.uu.nl/geobook/>.

[6] ERDöS, P. – RéNYI, A. On random graphs, I. *Publicationes Mathematicae (Debrecen)*. 1959, 6, s. 290–297. URL: <http://www.renyi.hu/~{}p_erdos/Erdos.html#1959-11>.

[7] MARECEK, J. et al. Dynamic Data Structures for Taskgraph Scheduling Policies with Applications in OpenCL Accelerators. In FOWLER, J. – KENDALL, G. –

MCCOLLUM, B. (Ed.) *In proceedings of the 5th Multidisciplinary International Conference on Scheduling : Theory and Applications (MISTA 2011), 9-11 August 2011, Phoenix, Arizona, USA*, s. 322–334, 2011. Paper.

[8] NIEMEIER, M. – WIESE, A. Scheduling with an Orthogonal Resource Constraint. In *10th Workshop on Approximation and Online Algorithms (WAOA2012)*, 2012.

[9] PAPADIMITRIOU, C. H. – TSITSIKLIS, J. N. On stochastic scheduling with in-tree precedence constraints. *SIAM J. Comput.* 1987, 16, 1, s. 1–6. ISSN 0097-5397. doi: 10.1137/0216001.

[10] TOBITA, T. – KASAHARA, H. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling.* 2002, 5, 5, s. 379–394. doi: 10.1002/jos.116. URL: <http://dx.doi.org/10.1002/jos.116>.

Appendix A

Terminology and symbols

Terminology

Task Execution unit without pre-emption, i.e. program function.

Kernel Task in OpenCL framework.

Taskgraph Directed acyclic graph where edges are formed by precedency constraints.

Makespan Time difference between start and finish of tasks scheduled.

Level In taskgraph, this goes by definition from [9] as "largest sum of expected processing times along a path in the dependency graph". In other words: Length of longest outgoing path from the node.

Depth Length of longest incoming path to the node.

SPP Strip Packing Problem

CLP Container Loading Problem

ADT Abstract Data Type

Gap Set of resources in Skyline structure

Rectilinear Polygon see *Gap*

Symbols

d	The number of resources and dimensions
r_i	Resource i capacity
t_j	Task j
T, U	Sets of tasks
p_j	Task processing time of task j
R_j	Set of resource constraints of task j
P_j	Set of precedence constraints on other tasks of task j
a_i^j	Array cell in dimension i in distance j from beginning
f	The number of object (i.e. gaps) facets
g	The number of gaps

Appendix B

Attached CD contents

```
+---bin                          executable
+---data                         test graphs and results
+---doc                          Doxygen documentation
|   \---html
+---src                          C++ source codes
+---thesis-kulovjir-2013.pdf     the thesis text
\---README.TXT                   this text
```

Figure B.1: Attached CD contents