LEARNING MODEL FOR CONSTRUCTION OF THE BEST DECISION SEQUENCE USING PRIOR KNOWLEDGE

by

Lilit Yenokyan

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Computer and Information Science)
In The University of Michigan - Dearborn
2009

Master's Committee:

Dr. Armen Zakarian, Chair
Dr. Pravansu Mohanty
Dr. David Yoon
Dr. Brahim Medjahed

Acknowledgments

First, I would like to express my deepest gratitude and appreciation to my academic advisor Dr. Zakarian for all his valuable advice and guidance during my Master's program. Thank you Dr. Zakarian for your exceptional attention, support and contribution of many hours and efforts into completion of this work. I have learned very valuable lessons and accomplished important milestones with your help.

I would like to thank Dr. Mohanty, Dr. Medjahed and Dr. Yoon for their support and valuable suggestions. It is a great honor that you agreed to be members of this thesis committee. I would like to express gratefulness to Dr. Grosky, Dr. Elenbogen and all the professors at Computer and Information Science Department for important lessons I have learned during my graduate program.

On a personal note, I would like to thank my parents for their unconditional love and encouragement thought the life. I will continue working hard to make you very proud. Finally, I thank my husband Viktor for his love, understanding and outstanding patience with my time constraints during these months of thesis work, it would be impossible to accomplish this work without your support.

Table of Contents

List of Figures

List of Tables

List of Appendices

Abstract

LEARNING MODEL FOR CONSTRUCTION OF THE BEST DECISION SEQUENCE USING PRIOR KNOWLEDGE

by

Lilit Yenokyan

Chair: William Grosky

The task of learning the best decision sequence based on different decision alternatives of accomplishing a goal is important in real life and technology. Assume a task is presented to a learning system, and steps taken to achieve an outcome are documented for all decision makers. Finding the best decision sequence to achieve the best possible outcome based on the prior decision instances is the learning objective. The goal of the thesis is to develop a system that is capable of learning from previous solution instances applied to the problem and able to construct a better solution to the problem using the learned knowledge. Initial approach of the proposed methodology examines various decision sequences for the purpose of learning. Next, the learning algorithm is developed to find a decision sequence which leads to the desired outcome. The approach presented in this thesis uses graph theory, learning algorithms and dynamic programming methods.

Chapter 1

Introduction

Machine learning is the ability of a computer to improve its problem solving capability based on previous results. The machine learning is a scientific discipline which is concerned with the development of theories and algorithms that allow computer technologies to learn from data collected from prior solution instances. Major focus of machine learning is to recognize complex patterns and make intelligent decisions analyzing those patterns. Many disciplines including artificial intelligence, data mining, neural networks and decision tree theory are concerned with the goal of investigating and finding new methods of machine learning.

The task of this thesis is development of a learning system capable of learning from prior solution instances of a problem and able of developing new solution procedures when similar problem is introduced to the system. This research considers particular type of learning system, where steps taken to solve the problem by various experts/decision makers (DM) is documented and stored in the system. Note that DM is an abstraction that may represent human, machine or system. Every step toward the goal presents an intermediate decision and outcome is a result of a combination of decisions. The only information available to the system to evaluate the correctness of decisions (actions) is the quality of the outcome.

Sutton and Barto (1998) define reinforcement learning (RL), as the learning what to do and how to map situations to actions to maximize a reward. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by merely trying them.

Our approach is based on fundamentals of reinforcement learning (Barreto and Anderson 2008, Whiteson and Stone 2006, Ribeiro 2002, Ghavamzadeh and Mahadevan 2007), which is often used for modeling real life applications such as, games (Fujita and Ishii 2007, Duan et al. 2007). The approach proposed in this thesis assumes that the information about the success or failure of an action sequence in problem solving is provided only when the sequence is complete. In contrary, each decision in RL problem solving is assigned a "reward" or "punishment" by the system depending on the correctness or incorrectness of the decision. Also, in RL process is accomplished through agents, while in the proposed approach graph theory is used for knowledge representation and inference. The similarity of developed model with RL is that both approaches use dynamic programming algorithms to find the best solution.

The thesis consists of two major sections. Initial approach builds a learning system to document the steps of decision sequences in problem solving and also information/data used for decision making. Directed Acyclic Graph[1] (DAG) (Stanley 1973, Bang-Jenson 2008, Czumaj et al. 2007, Pearce and Kelly 2006) is used to represent these decision sequences, where the nodes in the graph represent the actions taken by the problem solver and directed edges represent the order of the decisions (steps). It is assumed that multiple decision sequences are documented in the system using DAG, since in case of a single decision sequence documented in DAG no learning takes place and "learned" solution of the problem is the input to the system. As mentioned earlier typically many decision sequences are documented in the system and decisions that are based on the same arguments are mapped to the same

[1] In mathematics, a directed acyclic graph DAG, is a directed graph with no directed cycles; such that, for any vertex v, there is no nonempty directed path that starts and ends on v.

node in DAG. The algorithm presented in the thesis for the optimal DAG construction is analyzed in terms of computational time and storage space.

Second part of the thesis is concerned with finding the best solution from the learning space of DAG. The notion of “best” solution is based on the problem and the application (shortest completion time, shortest distance traveled, least load or cost required and so on). Dynamic programming approach (Bellman and Drayfus 1962, Bird and Moor 1993, Powell 2007) is proposed to quickly examine various solution alternatives documented in DAG and develop as new improved solution for the problem. Algorithm is developed to initialize the best solution sequence and it also serves as the basis of the recursion used in dynamic programming approach. Finally, new algorithm is developed that recursively searches for the best solution using dynamic programming approach.

The task of finding the diagnosis of a patient in medicine is presented in detail as application for developed approach. In this case entities are medical tests, decision makers are the doctors, and goal outcomes are diagnosis. Different doctors might come to same conclusions by performing different medical tests. In this case our goal is to learn based on diagnosis history of different doctors, and find the best sequence of tests, so that a minimum number of test are performed.

The situations that different sequences lead to same goal outcome are common in real life. Examples are not limited to medical field. Other application areas include road planning, manufacturing, product assembly, and so on. The examples are discussed in later section of the thesis after presentation of approach and algorithms.

Chapter 2

Literature Review

Focus of machine learning research is the construction of system that can automatically recognize knowledge patterns and make intelligent decisions based on prior accumulated knowledge. Learning process takes place whenever system under the consideration changes its structure or accumulates more knowledge (based on its inputs or in response to external information). The newly gathered knowledge then is used to improve future performance of the system. The improvements might be either enhancements to already performing systems or synthesis of new systems (Nilsson 1996). Machine learning systems are used in many scientific disciplines such as robotics, medical science or bioinformatics. They are also of increased interest in real-life applications such as, building personalized GPS navigation system or creating high-performance speech-recognition system.

2.1 Artificial Intelligence

The basic unit of intelligent system is the *agent*. An a*gent* is a computational mechanism that exhibits a high degree of autonomy and performs actions based on information (sensors, feedback) it receives from the environment it operates within (Panait and Luke 2005). Russell and Norvig (2003) define *agent* as anything that can sense, recognize and understand its environment through sensors and can act in that environment through effectors. AI based algorithms use the concept of *rational* agent in problem solving. Rational agent acts rationally and has high probability of making right decision at a moment of time. Several learning models (Goldman and Rosenschein 1996, Garland Alterman 2004, Haynes et al. 1996, Doniec et al. 2008, Montano et al. 2009) use *multi-agent* environment, where multiple agents actively

interact with one another to solve a common problem. To ensure effectiveness of the multi-agent environment, it is a constrained that at any given time an agent should not know *anything* about the world that other agents know (including the internal states of the other agents themselves).

Multi-agent systems (MAS), emphasizes the joint behaviors of agents that cooperate to solve a problem. MAS has two main learning models: cooperative and concurrent learning. *Cooperative learning* uses a team learning approach where all agents serve one common goal. In team learning system a single learner is involved, however this learner is discovering a set of behaviors for a team of agents, rather than a single agent (Haynes et al. 1996, Haynes and Sen 1995(a), (b), 1997). Although system involves a single learner it still poses challenges as agents interact with one another and the joint behavior can be unexpected. *Concurrent learning* uses multiple concurrent learning processes where each agent has main individual goal. Several learning processes attempt to improve parts of the team simultaneously in order to solve the proposed problem (Iba 1996, Iba 1998, Miconi 2003). Typically, each agent has its own unique learning process that modifies its behavior. In contrast to cooperative learning, concurrent learning approaches typically have a learner for each team member, assuming that this reduces the joint problem-solving space by projecting the problem into *N* separate spaces.

There are three main learning approaches in agent-based environment: *supervised*, *unsupervised*, and *reinforcement* learning. These methods are distinguished by type of feedback they provide to the learner. In supervised learning the correct output is provided. In unsupervised learning no feedback is provided. In reinforcement learning

the agent is rewarded for good responses and punished for bad ones (Russell and Norvig 2003).

Supervised learning includes elements of two theories: 1) classification – the ability of determining which category instance belongs to and 2) regression – based on set of input-output pairs develop function of input and output mapping. Due to complexity of interactions between multiple agents supervised learning methods are not easily applied to most problems as they assume that system provides agents with the "correct" behavior for the situation. The exception involving teaching in the context of mutual supervised learners is presented in (Garland and Alterman 2004, Goldman and Rosenschein 1996). Unsupervised learning has the ability to find patterns in a stream of input. The goal of unsupervised learning is determining how the data is organized. Most common forms of unsupervised learning include data clustering, Self-Organizing Map and Adaptive Resonance Theory models of neural networks. Reinforcement learning (RL) methods are especially useful in cases where reinforcement information (in terms of punishments and rewards) is provided after the actions take place in the environment (Barto et al. 1990, Kaelbling et al. 1996). RL methods are inspired by dynamic programming concepts and define formulas for updating the expected utilities and for using them for the exploration of the state space. Those methods have theoretical proofs of convergence. Along with the RL methods that estimate value functions there are *stochastic search* methods which directly learn behaviors without applying to value functions.

Semi-supervised learning is a type of learning that uses elements of both supervised and unsupervised learning models. It is a class of machine learning techniques that

uses both labeled (supervised learning) and unlabeled (unsupervised learning) data for training. Typically, a small amount of labeled data with a large amount of unlabeled data (Abney, S. 2008, Altun et al. 2005, Blum et al. 1998). Certain number of semi-supervised learning applications use graph based models (Zhu 2005 (a), (b)). Blum and Mitchell (1998) introduced combining labeled and unlabeled data with co-training and later introduced algorithms based on finding minimum cuts in graphs to learn from both labeled and unlabeled data (Blum and Mitchell 2001).

Recently there has been increased interest in non-centralized approaches to solving complex real-world problems. Many of these approaches use *distributed systems* technique where a number of entities cooperate towards achieving a common goal. The combination of AI and distributed systems led to distributed problem solving in which problem is decomposed and assign many processors or computers.

2.2 Decision Learning Trees

Decisions trees are used for data classification and learning based on attributes of data. Every branch in the decision tree represents a choice between a number of alternatives, and each leaf node represents a classification or decision. The values of the object attributes are propagated through the nodes of the tree to the leaf during object classification. Decision trees are used for classification learning. Each leaf in decision tree is a class outcome (not necessarily unique, as two distinct routs may lead to the same class). Learning stage involves creating new branches from root to the leaf with expected classification. After the learning stage is complete, upon arrival of new input without class data, it is propagated from root to leaf with most likely outcome. Thus larger learning data leads to more accurate classification and

unreliable probability estimates from small number of training instances often produce high error rates. Current-best-hypothesis approach used traditionally in decision tree learning has the disadvantage of using a single hypothesis and relying solely on probability for evaluating the generated inductive rules against input data**.** Version space method (Mitchell 1977, 1982), improves current-best hypothesis approach by maintaining the set of all consistent hypotheses and eliminating those found to be inconsistent with new examples. The approach was used in the expert system for chemistry *(Buchanan and Mitchell, 1978),* and later in LEX system (Mitchell 1983) to solve calculus problems. Decision tree algorithms, such as, ID3, C4.5 (Quinlan 1993, Murthy 1998), classification trees CN2 (Clark and Boswell 1989), are widely used in the industry. However they required large computing power and storage space and thus are limited to small scale problems. In order to address memory consumption issue most recent decision tree algorithms, such as, SLIQ, SPRINT (Han and Kamber 2006, Shafer et al. 1996), use presorting techniques to handle dataset that are too large to fit in memory.

2.3 Explanation Based Learning

Explanation Based Learning (EBL) is type of *analytic* learning that uses deductive mechanisms. Its advantage is that learning model of an example is sufficient for learning. In EBL one can use examples to directly solve new problems with analogical reasoning approach without any prior knowledge. To utilize the example the approach requires set of axioms about the domain of interest, goal concept and criteria for determining which features in the domain are efficiently recognizable. EBL has its bases in the techniques used by the automated STRIPS planner (Fikes et al. 1972). STRIPS uses divide and conquer approach to achieve conjunction of goals,

i.e., create a plan to achieve a preliminary goal and then create a plan to achieve the rest of the goals. Upon construction of a plan, its generalized version is saved in a plan library and used to support future planning as a macro-operator. EBL involves reformulating the domain theory to produce general rules that classify examples in a single inference step (Mitchell 1997). Classifying future examples that are similar to the training examples is performed very quickly. Recent analysis had led to a better understanding of the potential high costs of EBL in terms of problem-solving speed required for applying the learned rules (Minton 1988).

2.4 Relevance Based Learning

Relevance Based Learning (RBL) is a deductive learning approach that generalizes the information from prior (background) knowledge and uses hypothesis to create new learning examples. RBL information in the form of functional dependencies was first developed and is still mainly used in database community, where its main application is structuring large sets of attributes into manageable subsets (Russell and Norvig 2003). Tadepalli (1993) describes an ingenious algorithm for learning with determination that shows improvements in learning speed.

2.5 Inductive Logic Programming

Given the input set of examples represented as logical database of facts and encoding known to background knowledge, Inductive Logic programming (ILP) system can derive a hypothesized logic program which entails all positive and none of the negative examples. It is particularly useful in case of bioinformatics and natural language processing. ILP uses logic programming as a uniform representation of examples, background knowledge, and hypotheses. First major ILP program was

FOIL (Quinlan, 1990), which allowed practical induction of relational rules. Next major system introduced was CIGOL by Muggleton and Buntine (1988) that uses inverse resolution and capable of generating new predicates. More recent system PROGOL by Muggleton (1995) has been applied to a number of practical problems, particularly in biology and natural language processing. Muggleton (2000) describes an extension to PROGOL to handle uncertainty in the form of stochastic logic paradigms.

2.6 Neural Networks

Neural Networks (NN) were inspired and modeled based on brain neurons. It is a mathematical model that tries to simulate the structure aspects of biological neural networks. NNs consist of an interconnected group of artificial neurons and processes information using a connectionist approach to computation. They have wide range of applications, including in digital signal processing, detection of medical phenomena, stock market predictions, and so on.

The feed-forward neural network was the first and simplest type of artificial neural network devised. In this network, the information moves only in forward direction, from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network. The elementary type of the feed-forward network of only one node is perceptron. The back-propagation technique that is capable of adjusting its weights was introduced by Bryson and Ho (1969) but it was used after the publications (Werbos 1974, Parker 1985).

Contrary to feed-forward networks, Recurrent Neural Networks are models with bi-directional data flow. While a feed-forward network propagates data linearly from input to output, Recurrent Neural Networks also propagate data from later processing layers of the network to earlier layers. Hopfield networks (Hopfield 1982) are the most used class of recurrent networks. Hopfield network functions as an associative memory.

Other popular types of Neural Networks include *Support vector machines* and *Boltzman machines* (Vapnik 1998, Cristianini and Shawe-Taylor 2000, Scholkopf and Smola 2002). Support Vector machines are used in text categorization (Joachims, 2001), bioinformatics research (Brown et al., 2000), and text recognition (DeCoste and Scholkopf 2002).

NNs are statistical learning models, that are particularly useful for predictions and learning from continues data where most of discussed logical methods hardly succeed. The main drawback of the method is computational complexity and difficulty of finding the best cost function. Since our research deals with discreet data, where the learning system knows if it succeeded or failed logical programming methods such as modeling on graphs are more suitable and time efficient.

Chapter 3

Theory and Algorithms

3.1 Introduction

In this section theory and algorithms are presented for building learning system based on prior solution instances of a problem and developing new solution procedures when new problem instance is introduced to the system. The research focuses on specific type of learning system, where steps taken to solve the problem by various Decision Makers (DMs) are documented and stored in the system. DM is an abstraction that may present human, machine or system. Every step toward the goal presents a decision. This section focuses on concept and algorithms for creation of learning system and finding the best solution for the given instance.

First part of the section introduces the notations and algorithm used for construction of the knowledge base using the information extracted from decision sequences. The algorithm records decision sequences made by DMs. It examines the data and stores it in special type of Directed Acyclic Graph (DAG) (Stanley 1973, Bang-Jenson 2008, Czumaj et al. 2007, Pearce and Kelly 2006) with two distinct types of nodes (referred as *Hypertree* further in the thesis). The nodes in the Hypertree are used to present the decisions of the DMs and directed edges are used to present the decision sequences. The goal of knowledge construction algorithm is the construction of Hypertree that optimally stores the information from decision sequences, is easy to traverse and simple to enhance with the new decision sequences.

Second part of the chapter is devoted to finding the best solution from the knowledge base. Finding best solution means constructing a sequence in the Hypertree, which combines decisions made by the DMs that most efficiently leads to an outcome. Dynamic programming (Bellman and Drayfus 1962, Bird and Moor 1993, Powell 2007) approach is proposed to quickly examine various solution alternatives from the Hypertree and create a new improved solution. The construction of Best Decision Sequences (BDS) for list of outcomes (Algorithm 2) is used to build the best decision sequence for all the desired outcomes (Algorithm 3). The notion of "best" depends on the specific application, e.g. shortest time processed, shortest distance traveled, least load required, least cost required, least number of tests required, and so on.

3.2 Generating the Decision Knowledge Base

The goal of the knowledge base generation algorithm is to construct knowledge representation structure for decision sequences that lead to outcomes. The inputs of the algorithm are the sequences of decisions made by DMs. Each decision has set of attributes - parameters that describe the quality of the decision, such as cost, time and so on. The choices of the attributes may be large and mostly depends on the application area of interest.

Hypertree is a special hierarchical directed acyclic graph (DAG) which has two types of nodes, such as, *nodeSets* and *regular* nodes (see figure 1). NodeSet is a special node that contains list of references to set of regular nodes. Regular nodes are divided into following three categories: (1) basic entities (i.e., nodes with no incoming edges) represent initial decisions in decision sequences, (2) final entities (i.e., nodes with no outgoing edges) represent the final decisions in decision sequences and (3)

intermediate entities (i.e., nodes in the Hypertree with both outgoing and incoming edges) represent decisions leading from initial decisions to the final decisions.

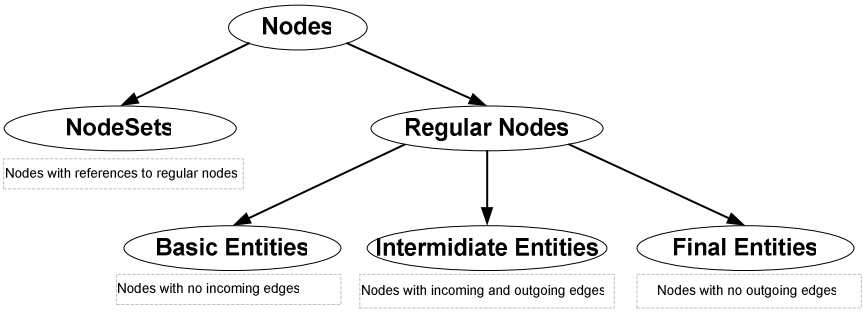


Figure 1: Hypertree Node Types

It is possible that various sequences produce same outcome with different attribute values. To capture the information on all decisions and attribute values that result to the outcome, the notion of *nodeSet* is introduced.

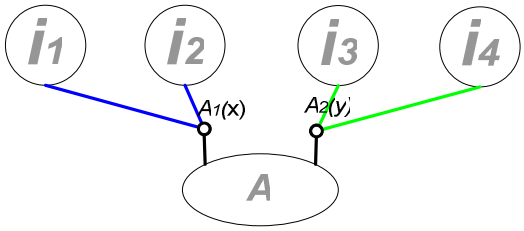


Figure 2: Example of two nodeSets resulting to outcome *A*

For example, in Figure 2, decision that results from merging nodes i_1 and i_2 is the outcome *A*, and so is the decision for merging i_3 and i_4. Therefore decision entity *A* can be obtained through two different decision sequences, through nodeSets A_1 or A_2. According to Figure 2, A_1 has references to nodes i_1 and i_2 and attribute value *x*, and A_2 has references to nodes i_3 and i_4 and attribute value *y*. The general structure of

nodeSet is such that it has references to list of nodes and attributes. All the intermediate and final entities in the Hypertree have one or more nodeSets. Notations of nodeSets from example in Figure 2 are presented as $A_1 = A^{\{x\}}_{\{i_1,i_2\}}$ and $A_2 = A^{\{y\}}_{\{i_3,i_4\}}$. Every node in Hypertree can also be presented by its nodeSets i.e. $A=\{A_1, A_2\}$, where A is intermediate or final entity.

Introduction of the concept of nodeSet allows to present knowledge base construction algorithm. Notations and the algorithm are presented next.

Notations:

$D^N_{A,\{e_1..e_n\}}\{a\}$ - Decision made by DM *N*, where A is resulting decision, $\{e_1, \ldots, e_n\}$ are the decision at the input that produce A, $\{a\}$ is the attribute value of the decision.

$DS^N = [D^N_{A,\{e_1..e_n\}}\{a\}, D^N_{B,\{e_c..e_n\}}\{b\}, \ldots, D^N_{M,\{e_g..e_h\}}\{m\}]$ - Decision sequence of decision maker *N*.

$A^{\{a\}}_{\{e_1...e_n\}}$ - NodeSet in the Hypertree, where A is resulting intermediate or final entity, $\{e_1, \ldots, e_n\}$ are the basic or intermediate entities at the input that produce A, $\{a\}$ is the attribute value of the nodeSet.

Algorithm 1 – Knowledge Base Construction Algorithm

Input: Set of decision sequences $DS = [DS^1, DS^2, \ldots, DS^S]$

Output: Hypertree of knowledge base *H_tree*

Step 1: Initialize *H_tree*=Ø

Step 2: **For** each decision sequence DS^m where $m = 1, 2,...,S$

For each decision $D^m_{A,\{e_1..e_n\}}\{a\}$ in decision sequence DS^m **do:**

Goto Step 3

Endfor

Endfor

Return *H_Tree*

Step 3: **For** each entity e_i *in* $\{e_1, \ldots, e_n\}$ **do:**

If e_i is not in *H_Tree* **then** add new node e_i to *H_tree*

Endfor

If *A* is not in *H_Tree* **then** add new node *A* to *H_tree*

Add new node set $A^{\{a_s\}}_{\{e_1,\ldots,e_n\}}$ to *H_tree*, with references to resulting node *{A}* and entity nodes $\{e_1, \ldots, e_n\}$ and attribute value *{a}*.

At the first step of the algorithm Hypertree *H_tree* is created and initialized as empty. Second step performs the traversal of all decision sequences, and for every decision in each decision sequence adds source and resulting decisions to *H_tree* as entities, if they were not already in *H_Tree* (step 3). In step 3, new nodeSet is created for every decision and added to the Hypertree. Once all the decision sequences are traversed and nodes added to *H_tree*, algorithm returns the constructed knowledge base in form of the Hypertree.

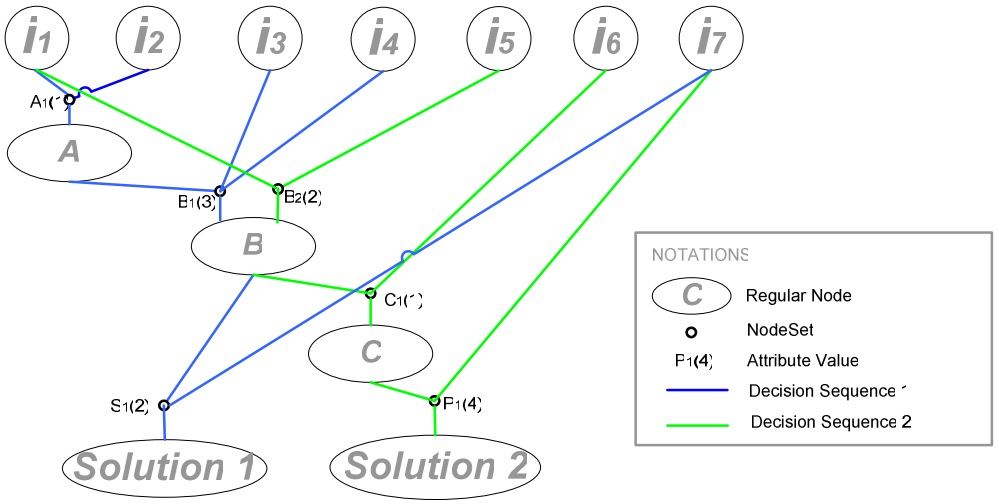


Figure 3: Knowledge base represented as Hypertree

Figure 3 presents the construction of Hypertree according to the algorithm 1 for two decision sequences shown in figure 3 with blue and green colors. After the Hypertree is initialized in Step 1, the algorithm traverses decision sequences and adds all the nodes and nodeSets resulting from the decision sequences to decision tree. It takes the first decision of the blue sequence $D^1_{A,\{i_1,i_2\}}\{1\}$, and adds a nodeSet $A_1 = A^{\{3\}}_{\{i_1,i_2\}}$, basic entities i_1, i_2 and intermediate entity A to the Hypertree. For the second decision $D^1_{B,\{A,i_3,i_4\}}\{3\}$, algorithm adds the nodeSet $B_1 = B^{\{3\}}_{\{A,i_3,i_4\}}$ to the Hypertree, along with basic entities i_3, i_4 and intermediate entity B, since intermediate entity A is already in the Hypertree. For third and final decision $D^1_{Solution1,\{B,i_7\}}\{2\}$ the same scenario is repeated and nodeSet $S_1 = Solution1^{\{2\}}_{\{B,i_7\}}$ added to Hypertree with the final entity *Solution1* and basic entity i_7. Once traversal of blue sequence is complete, new nodeSets $B_2 = B^{\{2\}}_{\{i_1,i_5\}}$, $C_1 = C^{\{1\}}_{\{B,i_6\}}$ and $S_1 = Solution2^{\{4\}}_{\{B,i_7\}}$ and corresponding entity nodes from the green sequence are added to the Hypertree. Note that after construction of decision Hypertree, node B is the only node that has two nodeSets $B=\{B_1, B_2\}$, as it is an intermediate node on both blue and green sequences. Once all

the decision sequences are traversed and all the nodes added to the Hypertree, knowledge base in form of the Hypertree is outputted from the algorithm.

3.3 Finding Best Decision Sequence from Knowledge Base

Once the knowledge base is constructed in form of Hypertree the goal is to find the best decision sequence with the best outcome from the knowledge base. Algorithms for constructing the best solution sequences are discussed in this section. Dynamic programming approach is used for examination of various solution alternatives from the Hypertree and construction of a new improved solution. The specific structure of the Hypertree allows building a recurrent expression (presented next), which is true for each node of the Hypertree.

$$BDS(A) = \min_{ns_i = 1,..,N} (a_i + \sum_{0<j<m} BDS(ns_i\{e_j\}) \quad [1],$$

where: *BDS* is Best Decision Sequence,

N is the number of nodeSets for the entity node *A*,

*N*odeSet ns_i is the *i-th* nodeSet of node *A*, a_i is the attribute value of nodeSet ns_i ,

$\{e_j\}$ is the *j*-th entity node in nodeSet ns_i resulting to *A*, and

m is the number of decisions sequences that result to *A* by nodeSet ns_i.

According to [1], the attribute value of the final entity is the sum of the attributes of the nodeSets in decision sequence that lead to that final entity. Therefore, it is important to know the nodeSet for every node that minimizes total sum of attributes for the node. Dynamic programming approach is used to avoid large number of recursive calls. It allows storing the values for each node that is already calculated and using that value for further recursive calls. In order to construct the best decision

sequence for the final node, the nodeSet index that results to a most favorable attribute value for each node in the sequence to the final node is stored in the system. Next, algorithms are presented for constructing BDS from the knowledge base represented as Hypertree using [1].

It is possible for a Hypertree to include multiple final entities of learning system. For example, decision sequences in figure 3. In general, the goal is to find the Best Decision Sequence for each final entity. Algorithm 2 is used to calculate the BDS for every final entity on its input. It initializes and executes Algorithm 3 for each final entity. Algorithm 3 computes BDS using [1], for each entity node of the Hypertree. The implementation of the recurrent expression in [1] uses dynamic programming (Algorithm 3) in order to store the values if entities (branches) in the Hypertree are already calculated.

Algorithm 2 – Construction of Best Decision Sequences for List of Outcomes

Input: Set of decisions to examine:

$$Out\_set = [O_{1\{ns_1 \ldots ns_n\}}, O_{2\{ns_i \ldots ns_j\}}, \ldots, O_{m\{ns_k, ns_h\}}]$$

Output: Best solution sequence for each final outcome in *Out_set*

Step 1: Initialize array of solution paths

Des_array[1,..,m] = Ø

Step 2: Calculate *BDS* for all the solution sequences

For each outcome sequence $O_{n\{ns_k \ldots ns_j\}}$ where *n = 1, 2,...,m*

Calculate and store *Des_array[n]=BDS(*$O_{n\{ns_k \ldots ns_j\}}$*)*

End For

Step 3: Return *Des_array[1,..,m]*

Algorithm 3 – Best Decision Sequence

Input: Node *A*

Output: Best Decision Sequence

Step 1: **If** *BDS(A)* has already been calculated **then** return *BDS(A)*

Step 2: Initialize:

current_value = 0

branch_value = 0

Step 3: **for each** NodeSet $A^{\{a\}}_{\{e_1,..e_n\}}$ **of** node *A* **do**

Set *branch_value* = $\{a\} + \sum_{i=1,..,n} BDS(e_i)$

If *branch_value* < *current_value* **then**

Set *current_value* = *branch_value*

Endif

Record index *k*, from $\{e_k\}$ which minimized *current_value*

Endfor

Step 4: **Return** *BDS(A)*

The application results of the algorithms 2 and 3 for Hypertree in figure 3 are presented in figure 4. Bold edges in Figure 4 are used to illustrate the Best Decision Sequences for the final decisions *Solution1* and *Solution 2*. Algorithm 2 is used to initialize and execute Algorithm 3 for decisions *Solution 1* and *Solution 2*. Here the attribute value for *Solution 1* is the sum of all the attributes of all the nodeSets that are included in the Best Decision Sequence shown on figure 4a. In this example the sum of all the attributes for *Solution 1* is equal to four. The result of algorithm 3 execution for *Solution 2* is shown in figure 4b, the sum of all attributes on BDS path is *7*. The BDS chosen for *Solution 2* corresponds to the decision sequence 2 (green sequence),

while BDS for *Solution 1* combines elements from both decision sequences (green and blue).

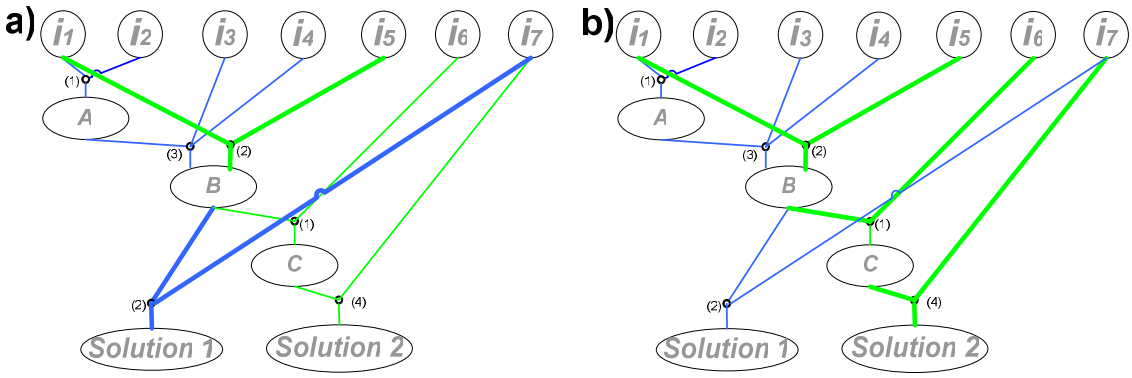


Figure 4: BDS execution for final entities Solution 1(a) and Solution 2(b)

Next BDS calculations (Algorithm 3) using [1] are presented for each final entity of the illustrative example shown in figure 5. Figure 5 is the extension of example shown in figure 3, with new decision sequence with only one decision resulting to *Solution 3*.

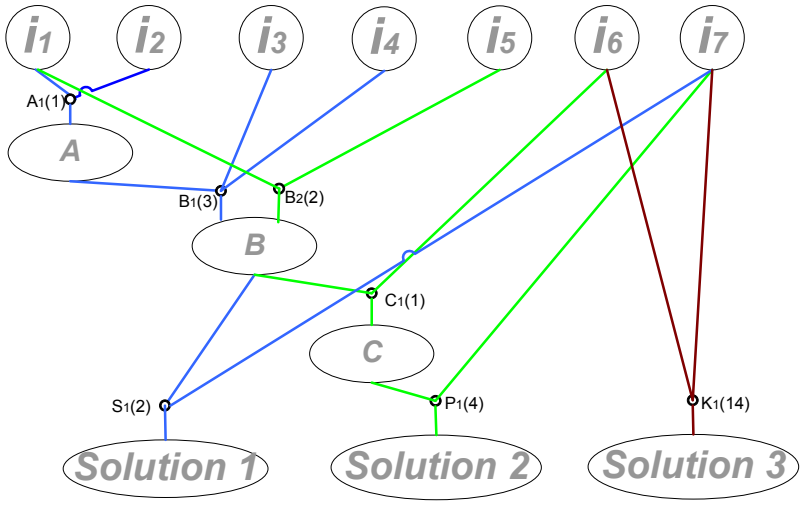


Figure 5: Hypertree in figure 3 enhanced with additional decision sequence

BDS calculation for the above examples is as follows:

$$BDS(Solution\ 3) = \min\{a_{K1} + i_6 + i_7\} = \min\{a_{k1}\} = 14$$

Since the basic entities do not have NodeSets and attributes, the values of i_6 and i_7 are 0, and the BDS for *Solution 3* equals to the value of attribute of its only nodeSet. Therefore:

$$BDS(Solution1) = \min\{a_{S1} + BDS(B) + i7\} = \min\begin{Bmatrix} a_{S1} + [a_{B1} + BDS(A) + i_3 + i_4] \\ a_{S1} + [a_{B2} + i_1 + i_5] \end{Bmatrix} =$$

$$= \min\begin{Bmatrix} a_{S1} + a_{B1} + [a_{A1} + i_1 + i_2] + i_3 + i_4 \\ a_{S1} + a_{B2} + i_1 + i_5 \end{Bmatrix} = \min\begin{Bmatrix} 2+3+1 & =7 \\ 2+2 & =4 \end{Bmatrix} = 4$$

BDS for Solution 1 is computed by comparing the attribute value of two decision sequences (i.e., blue and green sequences in Figure 5) and selecting the sequence with the most favorable attribute value. Minimum attribute value is *4*, which improves its initial attribute value of *6*. The *BDS* calculation for *Solution 2* is presented next:

$$BDS(Solution\,2) = \min\{a_{P1} + BDS(C) + i_7\} = \min\{a_{P1} + [a_{C1} + BDS(B) + i_6] + i_7\} =$$

$$= \min\begin{Bmatrix} a_{P1} + a_{C1} + [a_{B1} + BDS(A) + i_3 + i_4] + i_6 + i_7 \\ a_{P1} + a_{C1} + [a_{B2} + i_1 + i_5] + i_6 + i_7 \end{Bmatrix} =$$

$$= \min\begin{Bmatrix} a_{P1} + a_{C1} + a_{B1} + [a_{A1} + i_1 + i_2] + i_3 + i_4 + i_6 + i_7 \\ a_{P1} + a_{C1} + a_{B2} + i_1 + i_5 + i_6 + i_7 \end{Bmatrix} = \min\begin{Bmatrix} 4+1+3+1 & =9 \\ 4+1+2 & =7 \end{Bmatrix} = 7$$

BDS calculation for *Solution 2* again compares the value of two decision sequences (i.e., blue and green sequences in Figure 5) and selecting the sequence with the most favorable attribute value. Here the initial attribute value was *7* is not by *BDS* calculation.

Chapter 4

Applications and Examples

4.1 Introduction

Learning systems have application areas in many disciplines of science and technology. The system proposed in this thesis builds a knowledge base that captures all the decision sequences for various decision makers (DMs) that produce various outcomes. The objective of the system is to learn from prior decision sequences, and produce a sequence that has most favorable value for the specified outcome. Note that best decision sequence is either one of the sequences presented by the decision makers or a new sequence derived from multiple decision sequences. The learning model proposed in this thesis is suitable for problems where solutions are achieved by decision sequences presented by experts, and goal of the model is to improve quality of the sub problem or the entire problem.

One application area may be establishing patient diagnosis from healthcare. The goal here is to identify the best sequence of tests for developing accurate diagnosis of patients with certain symptoms. Various sequences of tests prescribed by doctors for patients with similar symptoms and their final diagnoses are captured in the knowledge base. The goal of learning system is to accumulate knowledge from those sequences and produce a sequence of decisions (tests) for establishing diagnoses for a patient. The sequence is build based on specified criteria, such as cost, accuracy, lead time to diagnose.

Another application area discussed in this section is product assembly. In product assembly application various feasible assembly (decision) sequences are captured in a knowledge base. The goal of the learning system is to learn from those product assembly sequences and produce a sequence that is optimal with respect to some specific criteria (i.e., assembly time, cost, number of setups, and so on).

Assisting drivers finding best driving directions between the origin and destination is another possible application area for developed approach. Here various driving routes between the origin and destination are stored in the knowledge base. Goal of the algorithm is to examine those routes and produce a new path that is optimal with respect to specific criteria, such as time, distance traveled, and so on.

Examples presented in this section include demonstration of the developed approach for healthcare and product assembly applications.

4.2 Medical application

An application proven from healthcare, such as establishing patient diagnosis using prior similar patient diagnostic instances is discussed in more detain next. In this application example, initial entities in the knowledge base model are medical tests, intermediate entities are the decisions of doctors based on those tests, and final entities are final diagnoses. Study involves medical history of group of patients with the similar initial symptoms and complains.

The goal of the system is to learn from prior diagnostic history of patients, and be able to develop a diagnostic plan for a new patient with similar or overlapping symptoms.

This plan will include set of intermediate decisions (tests) with the objective of establishing final accurate diagnosis. Also the goal of the system is to develop a plan which has favorable value with respect to some attribute, e.g. cost, time to diagnose, least number of tests, and so on. Assuming that the level of proficiency is same for all the doctors, they all will come to the same conclusion after some point for patients with similar history and complains.

To illustrate the application of the developed approach, consider the diagnostic history of patients with similar symptoms. The history of symptoms and diagnoses are documented for the patients and later the patients are divided by a medical professional to groups of people who had similar symptoms. This grouping criteria helps to find which the most probable diagnoses are for a new patient just added to the group, also helps doctor to establish the optimal set of tests that should be prescribed for examining the patient for any of the probable diagnoses in the group. During the examination of a patient from a group, every test prescribed by the doctor is recorded, doctor also supplies the system all the possible conclusions or intermediate decisions he/she makes based on results of each test.

Knowledge capture algorithm documents not only sequences of tests but also attributes of the tests, such as cost and test lead time. To form a single decision sequence, the system records every test (initial entities), intermediate results and decisions (intermediate entities) and final diagnosis (final entity) for each patient. The knowledge base grows once more records are captured in the learning system. It is important to note that the intermediate outcomes and final diagnoses can be reached by performing various combinations of tests. After a number of patients with same set

of symptoms are examined, and sufficient knowledge base is formed. The BDS algorithm is then used to find a most favorable set of tests for establishing the diagnosis using a specified attribute.

The general example of the application scenario is presented further. Every colored line on figure 6 presents the series of tests performed during the examination of a patient. $\{t_1, t_2, ..., t_n\}$ represents the set of tests prescribed by doctors in order to diagnose a patient from specific group, those tests present the initial entities in the knowledge base. The intermediate decisions and tests ordered by doctors are noted as $\{ID1, ID2, ...\}$. Finally the diagnoses or outcomes of the algorithm are noted as $\{Diagnose\ 1, Diagnose\ 2, ..., Diagnose\ m\}$. The value of performing some tests after having done set of tests is shown as attribute of the nodeSet of joining those tests. The attribute value can be measured in terms of cost or lead time for scheduling and performing tests.

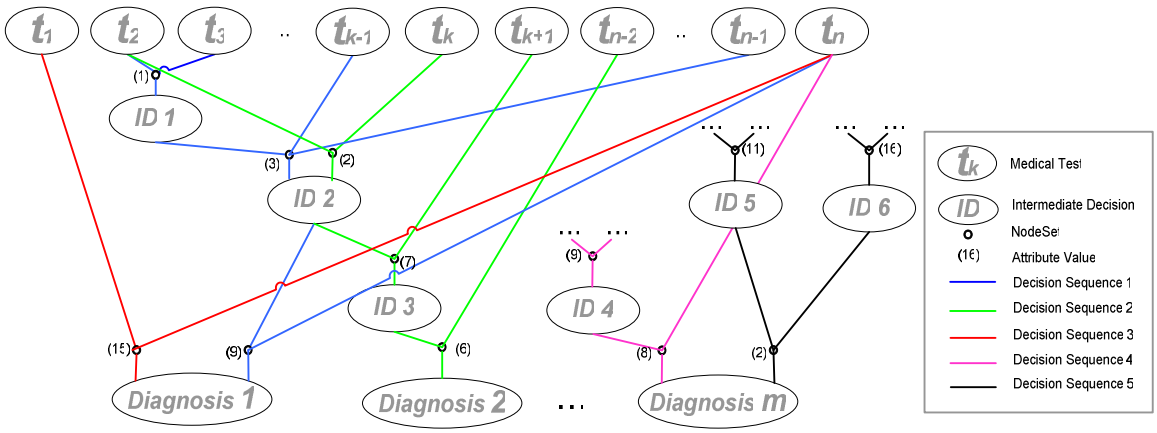


Figure 6: Illustration of Medical application

Figure 6 is the result of running knowledge base construction algorithm on all decision sequences available to the systems. It constructs the knowledge base from the decisions sequences of diagnoses of various patients. Since in medical application the intervention of medical professional is crucial, the learning system is used as advisory to make decisions and can not substitute human role.

Construction of BDSs for list of outcomes (Algorithm 2) finds the minimum value and best sequence for final entities given at its input. In medical application example for a patient the Algorithm 2 is executed for all the potential diagnoses that doctor finds likely from the set of diagnoses that were proven for patients with similar symptoms. For illustration proposes suppose that for a certain patient the medical professional found that *Diagnosis 1* and *Diagnosis 2* from figure 6 are the most likely. Algorithm 2 is used to create the Best Decision Sequence for *Diagnosis 1* and *Diagnosis 2*. Resulting BDSs shown on figure 7.

For *Diagnosis* 1, there are two possible decision sequences in the knowledge base to establish correct diagnosis for a patient, denoted by red and blue branches in figure 6. Since those sequences do not have any common intermediate decisions, using BDS algorithm the sequence that has the most favorable value for a specified outcome is returned. Attribute value of a decision sequence represented by the red branch is *15*, and it is constructed in a single step, hence finding the best construction sequence for blue path presents more challenge. The initial attribute value of blue branch is *(1+3+9=13)* however the intermediate entity *ID1* is used during construction of both blue and green sequences. Therefore the best decomposition of *ID1* should be chosen between sequences corresponding to blue and green branches, even though green

branch leads to different diagnose then *Diagnosis 1*. Attribute value of a decision sequence represented by the green branch *(2)* is smaller then that of the decision sequence represented by the blue branch *(4)* for *ID1* decomposition, thus it is chosen by BDS algorithm, resulting to figure 7a for *Diagnosis 1*. Since there is only one sequence that results to *Diagnosis 2*, only the attribute value of intermediate results of sequence construction can be optimized. The only intermediate result for the green branch that is common with other sequences is *ID2* which was decomposed already. The BDS for *Diagnosis 2* is the green decision sequence, as it results to the smallest total of all test values *(15)*, presented in figure 7b.

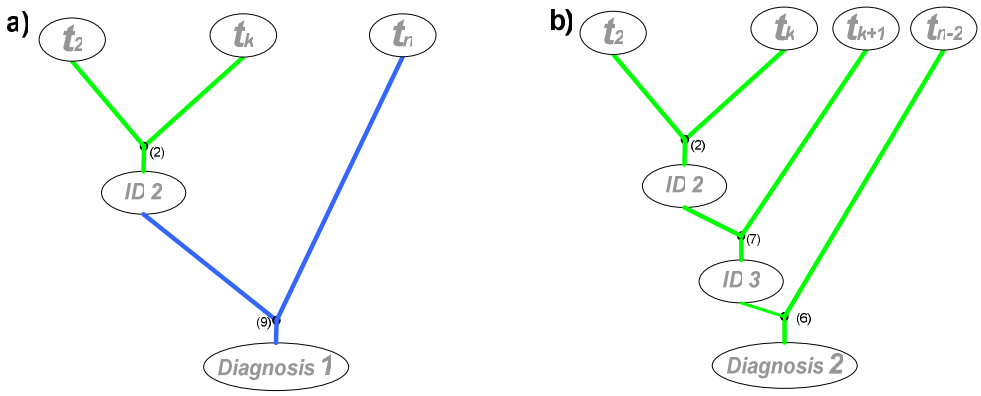


Figure 7: BDS constructed for Diagnose 1 (a) Diagnose 2 (b).

Comparisons of attribute values (i.e. costs) for establishing *Diagnosis 1* is *11* to initial costs *15* (red decision sequence) and *13* (blue decision sequence). Therefore this is an improvement of 15% percent of the initial cost. The attribute value for establishing *Diagnosis 2* is the same as it was initially *15*.

When the BDS path is constructed for a Diagnosis, the tests should be done as indicated in the sequence of BDS. At any step if a test results do not satisfy the

condition and diagnose *k* should not be considered any longer, sequence for performing the tests stops, as negative decision about *diagnosis k* has been reached.

If during patient inspection doctor decides to examine a diagnosis that is not yet in the knowledge base (i.e. for the knowledge base with final decisions *Diagnosis 1* and *Diagnosis 2* on figure 6 an new final decision *Diagnosis 3* is introduced) the system can not construct a BDS to reach the new diagnosis since there is not enough information how the diagnosis was reached. Instead, the learning system records new diagnosis and the tests prescribed by the doctor in the knowledge base, in order to assist the doctor for examining the *Diagnosis 3* in the future.

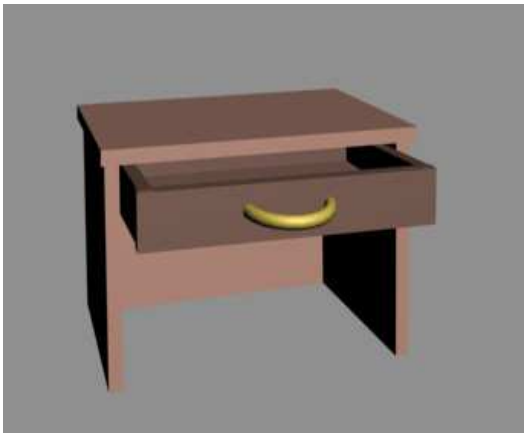
4.3 Product Assembly Application

In this example finding optimal product assembly sequence is considered. Assembly of any size from furniture to aircrafts is among possible applications of developed concept.

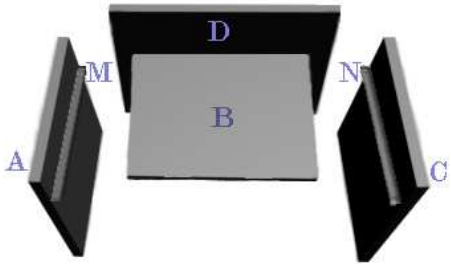
The focus of this chapter is illustrating the optimization algorithm on a relatively simple example of manual assembly of the desk. Given all the necessary mechanical parts of the desk, a number of professionals are assigned to independently assemble it. The professionals perform the assembly in various orders requiring various amounts of time. The assumption is that each professional has same level of competency and that the time required performing a certain task is the same for all of them. The time of each action performed is available, i.e. system is tracking the time it takes the professional to perform each step of assembly. Having several assembly sequences, the goal is to construct the best sequence with minimum total duration.

Figure 8a is the 3D render of the of desk object considered in this example. Desk has total of 12 components and two major subassemblies, namely the base (figure 8b) and the drawer (figure 8c). Components in figure 8 are numbered (*A, B, C ...).* To complete the assembly the drawer should be attached to the base through parts *H* and *M*, and parts *F* and *N*. However, it is clear that not any sequence of attachments of parts may result to the correct desk assembly (details in appendix B).

a)



b)



c)

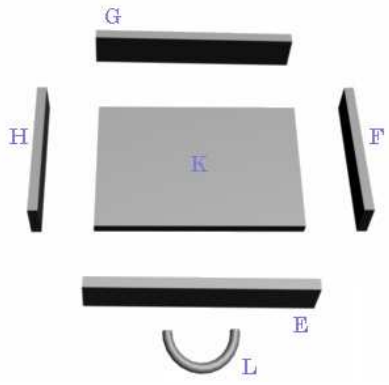


Figure 8: Desk example – (a) 3D rendering of assembled desk view
(b) Components of the base part of the desk (c) Components of the drawer

Knowledge capture algorithm is used to store all the existing sequences of the desk assembly. Operation in the context of this discussion implies assembling several parts of the desk together. Since it is assumed that only one individual performs the assembly, therefore only two parts can be assembled together at any given time. Basic

entities in the knowledge base are the parts of the final product. Intermediate entities are subassemblies obtained by sequence of operations, and the final entity is the assembled desk. The specific feature of assembly application is that there is only one final outcome – the assembled desk, and all the parts of the desk (basic entities) must be assembled together to produce the final desk. The time required to assemble several parts together is the attribute of the operation. After having sufficient knowledge base, i.e., having examined several of assembly sequences, BDS algorithm is used to construct the best assembly sequence to complete assembly in the shortest time.

Three assembly sequences of desk assembly example are shown in Table 1:

a) RED LINE	**b)** GREEN LINE	**c)** BLUE LINE
1) **A, B → AB (2)** *2)* **C, AB → ABC (2)** *3)* **D, ABC → ABCD (4)** *4)* **K, F → KF (3)** *5)* **G, KF → KFG (6)** *6)* **H, GKF → KFGH (6)** *7)* **L, E → EL (5)** *8)* **EL, KFGH → Drawer (5)** *9)* **M, ABCD → ABCDM (7)** *10)* **N, ABCDM → Table (7)** *11)* **Table, Drawer → Desk (6)**	*1)* **A, M → AM (2)** *2)* **C, N → CN (2)** *3)* **AM, B → ABM (2)** *4)* **CN, ABM → ABCNM (2)** *5)* **D, ABCNM → Table (5)** *6)* **F, K → KF (3)** *7)* **H, FK → KFH (3)** *8)* **G, KFH → KFGH (5)** *9)* **E, KFGH → Drawer_no_handle (2)** *10)* **Drawer_no_handle, Table → Desk_no_handle (6)** *11)* **L, Desk_no_handle → Desk (14)**	*1)* **A, M → AM (2)** *2)* **C, N → CN (2)** *3)* **D, AM → ADM (2)** *4)* **ADM, CN → ACDNM (3)** *5)* **B, ACDNM → Table (1)** *6)* **F, K → KF (3)** *7)* **G, KF → KFG (6)** *8)* **H, KFG → KFGH (6)** *9)* **L, E → EL (5)** *10)* **Table, KFGH → Desk_no_front (4)** *11)* **Desk_no_front, EL → Desk (11)**
Total: 53 minutes	Total: 47 minutes	Total: 45 minutes

Table 1: Decision sequences of desk assembly

Each row of each decision sequences on table 1 is an operation. The first operation of sequence (table 1a) *A, B → AB (2)* indicates that elementary parts *A* and *B* are assembled together producing subassembly *AB* with processing time of 2 minutes. The second operation *C, AB → ABC (2)* indicates that elementary part *C* is joined with subassembly *AB* producing subassembly *ABC* in two minutes. Introduced syntax assumes that only parts and subassemblies can occur at the left of "→" sign, and only

subassemblies and final product assembly notation can occur at right from the sign. The subassembly at the right of the expression can be given arbitrary name, such as at step 8 of the red sequence *EL, KFGH → Drawer (5)*. Decision sequence marked by red color completes when two subassemblies *Drawer* and *Table* are assembled together producing *Desk* in 6 minutes, i.e. *Drawer, Table → Desk (6).*

Decision sequence from table 1(a) completes the execution in total duration of 53 minutes, decision sequence from table 1(b) requires total of 47 minutes and decision sequence 1(c) requires total of 45 minutes. The goal is constructing a decision sequence that has the least total duration based on the three decision sequences. The duration of best decision sequence should be less or equal the minimum duration of all three decision sequences. In case that there were no common subassemblies between the decision sequences the solution would be the decision sequence that takes the minimum time, in this case blue sequence 1(c). But there are common subassemblies such *Table* (ABCDNM) resulting from operation 10 of decision sequence 1(a), operation 5 of decision sequences 1(b) and 1(c), they all construct the base part of the desk without the drawer.

The sequence of red decision sequence (table 1a) is shown on figure 9. The numbers at the right indicate the “level” of assembly that is the number of elementary parts assembled together to form that subassembly. For the elementary parts the level is 1 and for the assembled desk the level is 12.

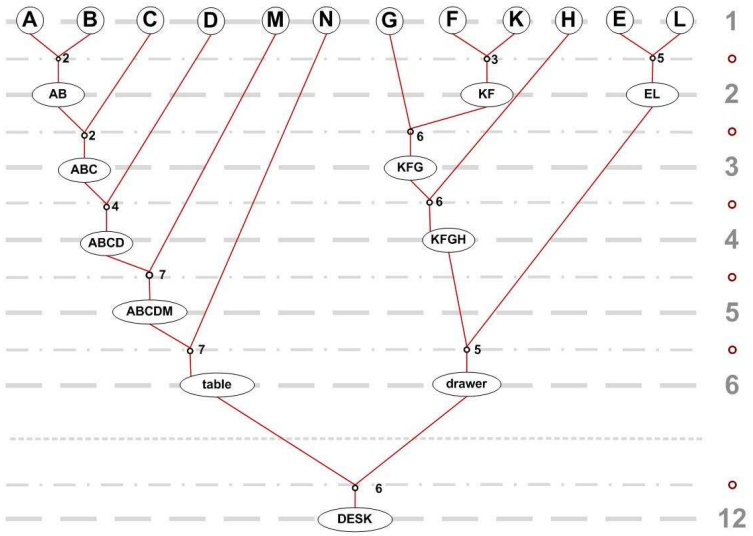


Figure 9: Assembly of decision sequence 1(a)

Graph illustration of the green decision sequence (table 1b) is shown in figure 10, and graph illustration of blue sequence (table 1c) in figure 11.

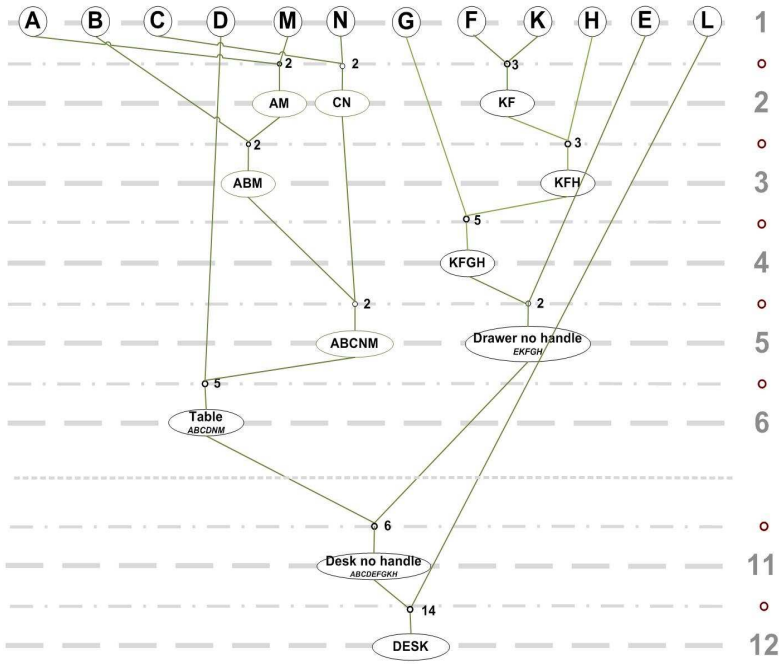


Figure 10: Assembly of decision sequence 1(b)

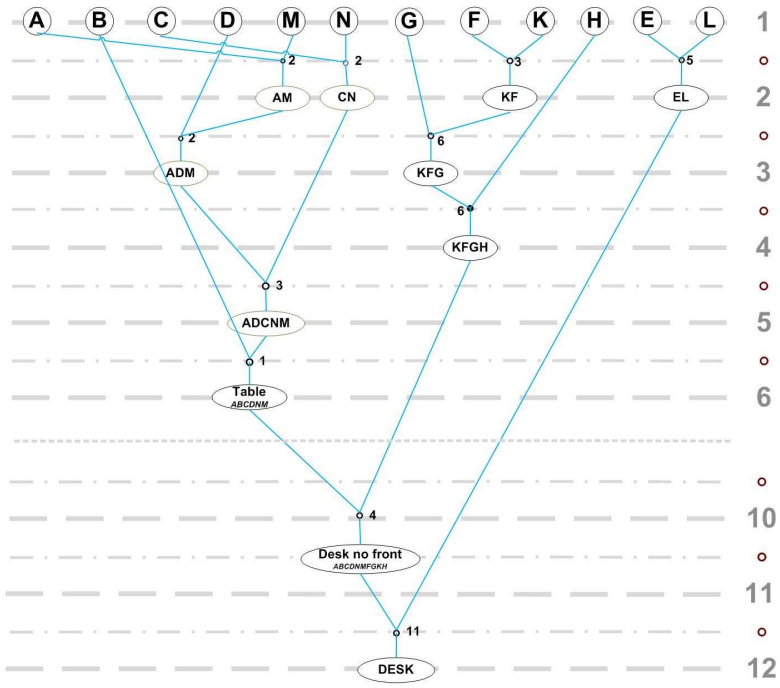


Figure 11: Assembly of decision sequence 1(c)

The result of knowledge base construction algorithm for decision sequences from table 1 is displayed in figure 12. Decision sequences 1(a), 1(b), and 1(c) are marked in red, green and blue colors respectively.

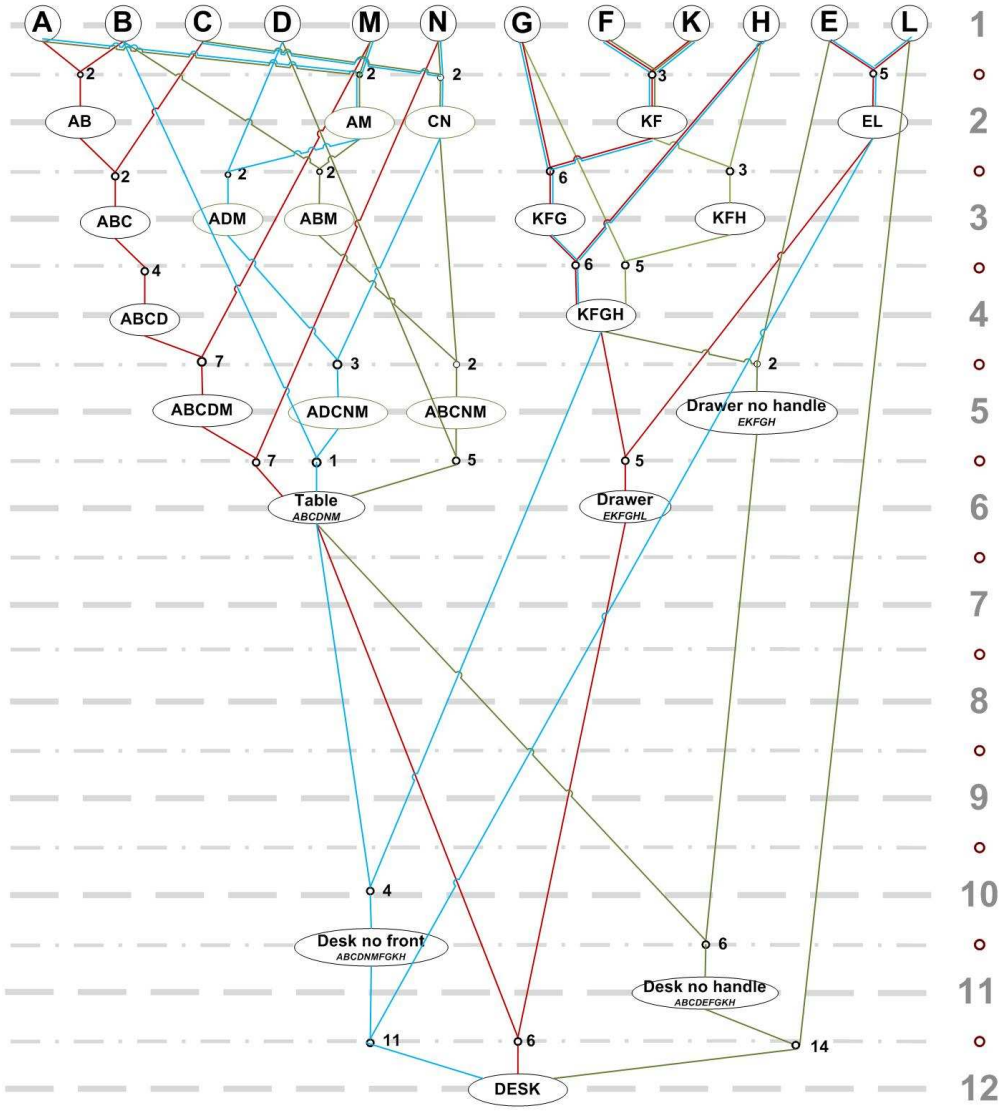


Figure 12: Knowledge base for desk assembly

In the knowledge base several nodes have multiple NodeSets, such as final entity "Desk" as it is the result for final decision in each of decision sequences. In figure 12 the edges are marked with multiple colors when several decision sequences construct same node using the same operation. Once the knowledge base is constructed the BDS algorithm is executed to construct best decision sequence, one that takes least amount of time to assemble the desk. The resulting sequence is presented in figure 13.

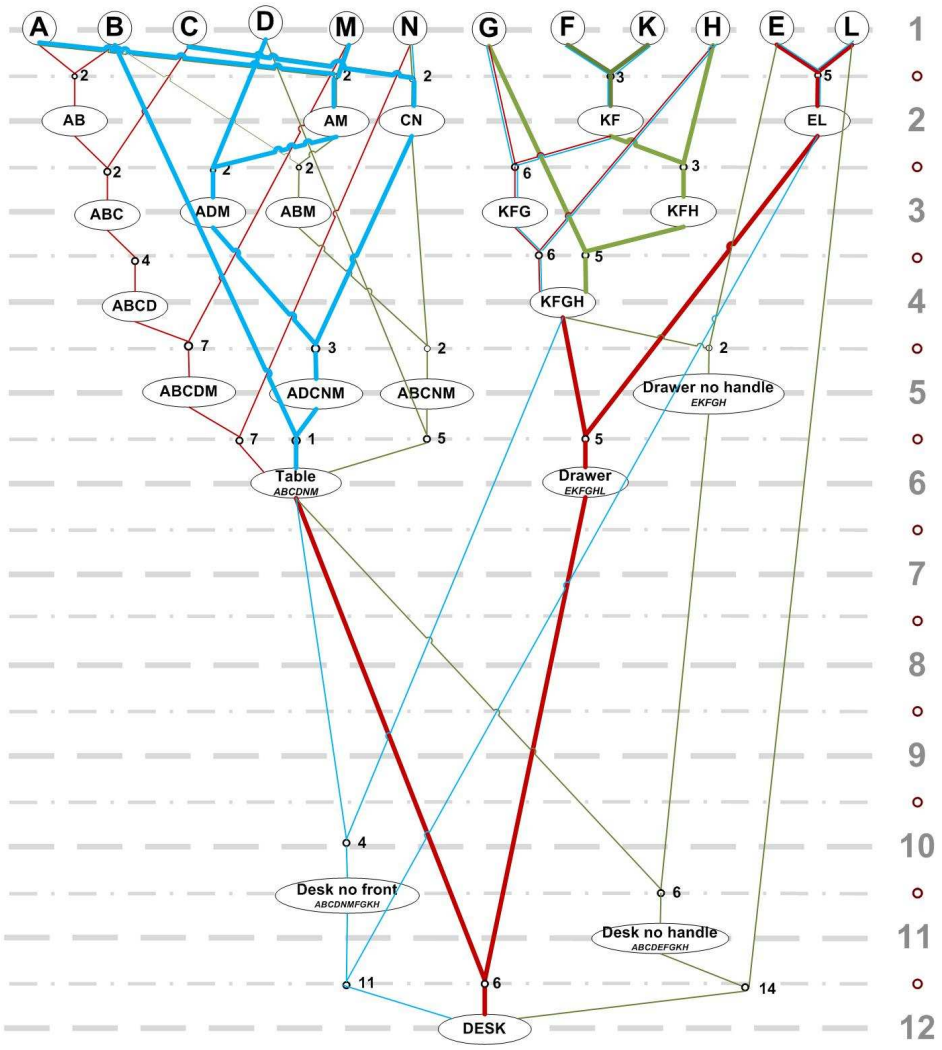


Figure 13: BDS for Desk assembly

Bold edges in figure 13 correspond to operations in the optimal assembly (BDS). The operations from all three existing decision sequences in table 1 are used to form the best assembly sequence of the desk. Total amount of time required for assembling the desk based on existing knowledge is 37 minutes which is 17% improvement from the previous shortest assembly sequence from table 1(c) of 45 minutes. The final assembly sequence is presented in table 2:

blue	***1)*** **A, M → AM (2)**
blue	***2)*** **C, N → CN (2)**
blue	***3)*** **D, AM → ADM (2)**
blue	***4)*** **ADM, CN → ACDNM (3)**
blue	***5)*** **B, ACDNM → Table (1)**
green	***6)*** **F, K → KF (3)**
green	***7)*** **H, FK → KFH (3)**
green	***8)*** **G, KFH → KFGH (5)**
red	***9)*** **L, E → EL (5)**
red	***10)*** **EL, KFGH → Drawer (5)**
red	***11)*** **Table, Drawer → desk (6)**
Total: 37 minutes	

Table 2: Best decision sequence for desk assembly

The assembly application was illustrated for example of desk assembly. Larger size assembly projects, such as houses or airplanes are among possible applications for system developed in this thesis.

Chapter 5

Conclusion

The task of learning the best decision sequence based on different decision alternatives of accomplishing a goal was discussed in this thesis. Development of a system that is capable to learn finding the best solution from previous solution instances applied to the problem was the goal of the thesis. First an approach was proposed to examine various decision sequences for the purpose of learning and create a knowledge base based on those sequences. Then algorithms were developed to find a decision sequences which lead to the desired outcomes. Finally the illustration of the developed approach was discussed in detail for two real life examples from healthcare and product assembly. Application areas for developed concept are numerous, medical tests and desk assembly were two possible examples, chosen for relative straightforward nature. The developed approach in this thesis combined elements of graph theory, learning algorithms and dynamic programming methods.

Appendices

Appendix A

Graph of the rules

As part of implementation of the algorithms a software tool was developed for the specific example of the product assembly. Given the sketch of the assembly, the software generates the “graph of rules” of all the parts that should be joined together. In the graph of rules two vertexes have a common edge if the two parts are supposed to be attached together.

a) b)

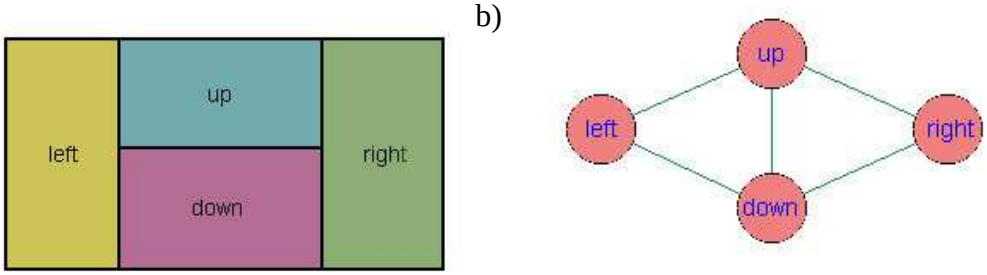


Figure 14: Graph generation (a) Sketch of the mechanical object.
(b) Graph of the rules.

Simple example rule graph generation is shown in figure 14, based on the input of the sketch of mechanical object consisting of four parts (up, down, left and right) (figure 14a) the graph of the rules is generated (figure 14b). If the parts on the sketch are adjacent, that means those parts should be joined together, represented by common edge in the graph of the rules. The information from graph of the rules is used later in the execution of the algorithms, to restrict illegal attempts of joining parts together that do not supposed to be joined, e.g. parts *left* and *right* from figure 14.

Appendix B

Deadlock elimination

In assembly examples containing large number of parts in case of some incorrect assembly sequences situations leading to dead ends (called deadlocks further) are possible. These situations occur in case several parts are attached together correctly, which however makes impossible to attach required parts further, a example of concept is shown on figure15.

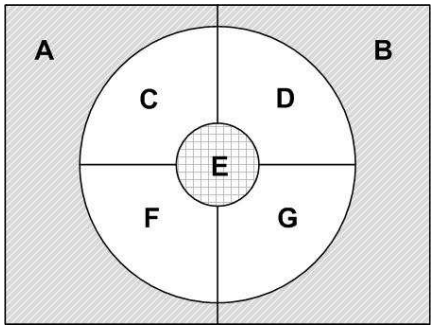


Figure 15: Example of double deadlock

An example of geometric limitations in three-dimensional presentation of objects is illustrated in figure 15. Assume that there are geometric limitations similar to figure 15. The parts *C, D, F* and *G* are attached together at first it makes impossible to attach part *E* without detaching the initially attached parts first. Moreover, if the first action was attachment of parts *A* and *B*, it will make unfeasible to attach to the already constructed part any of the parts *C, D, F, G* or *E.*

a) b)

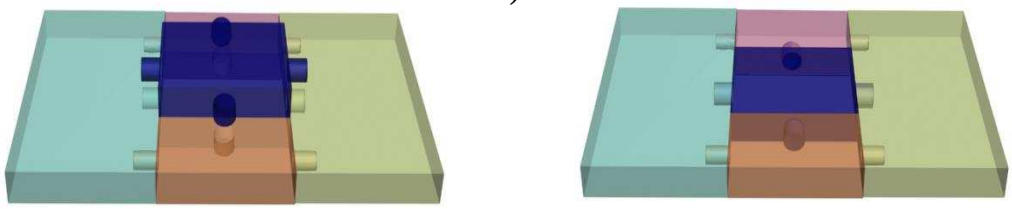


Figure 16: Deadlock based on geometric limitations.
(a) Occurrence of deadlock (b) Merged deadlock

The assembly decision sequence can start from attaching any two parts, and continue which might lead to the deadlock situation. The strategy of handling the deadlock occurrences is first finding the innermost part that cause deadlock and requiring that it needs to be attached to other parts first. Then same action needs to be repeated for all the inner parts causing deadlock, until the situation is resolved. These results to making the rules graph "partially directed" graphs, where directed edges show the required order of precedence of attachments to avoid deadlocks. Therefore the rules graph generated based on sketch on figure 15, will be figure 17 (b), not 17 (a).

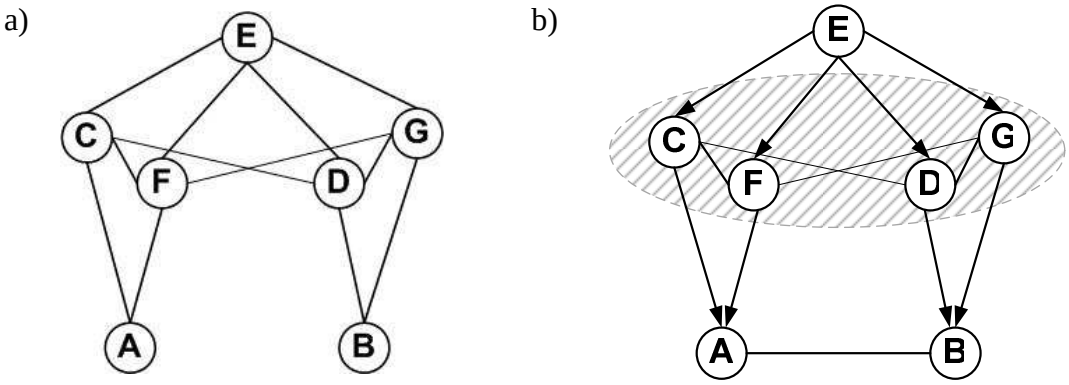


Figure 17: Rules graph generated based on figure 15. (a) Rules graph with undirected edges (b) Rules graph with combined directed and undirected edges

In figure 17 a there are no directed edge which means any order of attachment is possible. To avoid occurrence of deadlocks partially directed graph on figure 17b is constructed. Directed edges in figure 17b present the order of precedence of actions to avoid deadlocks. This means that if there is an incoming directed edge for a vertex, the part corresponding to the source node of directed edge should be attached to its destination, before the destination node is attached to any other node. E.g. part *G* cannot be attached to any other part, unless part *E* is attached to it. Parts *A* and *B* can be attached together only if they are attached to all the other parts. Thus attachment of *A* and *B* should be the last action.

In the main example of desk assembly there is a possibility of deadlock occurrence. Recall figure 8c – assembly of the drawer. In case the sides of the drawer are attached together, it is impossible to attach the drawer bottom. That is if parts *H*, *G*, *F* and *E* are attached there is no way to attach *K*, without first disassembling the rest of drawer.

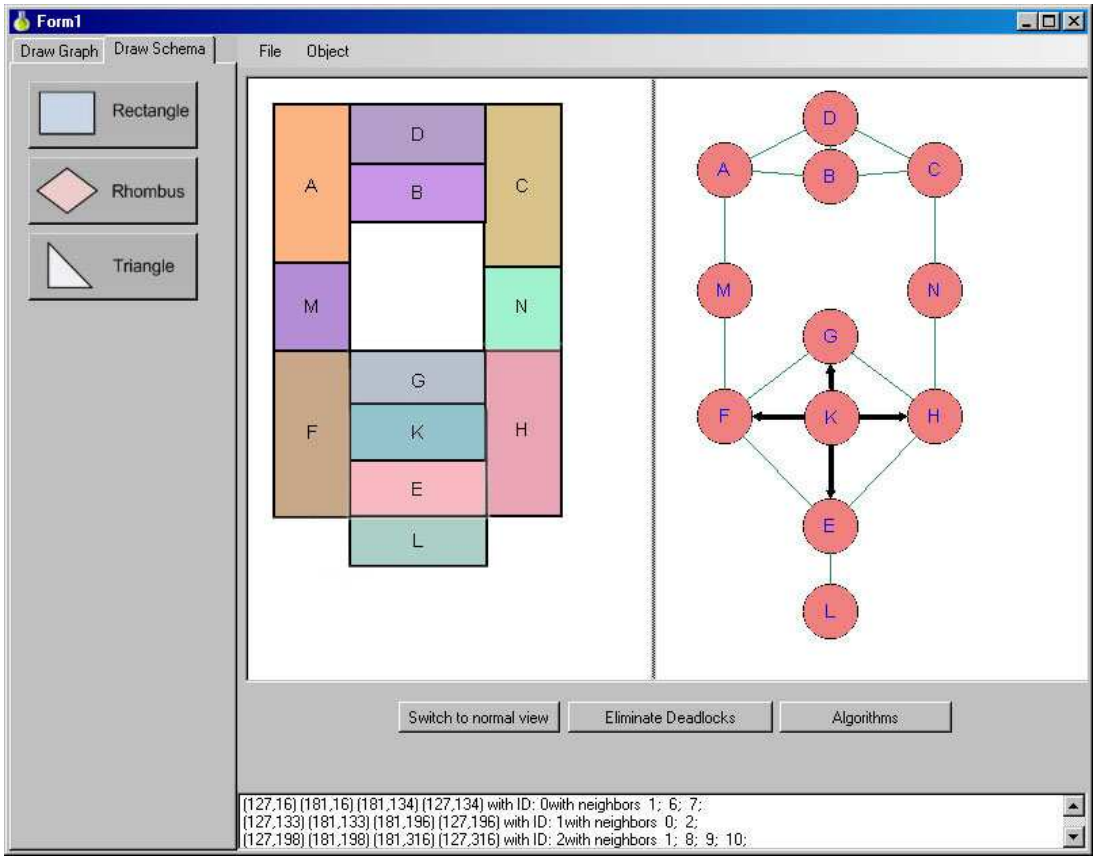


Figure 18: Screenshot of the desk assembly rules generation

At the central part of the program execution screenshot on figure 18 the schema (sketch) of desk assembly is shown based on its parts form figure 8. At the right the generated rules graph is shown. The names of the vertexes correspond to the names of the parts of the graph. Vertexes have common edge if the parts have to be attached together (*N* and *M* parts are the rails that attach drawer to the desk). The rules graph has undirected edges where the order of joining parts is not important. Therefore graph has undirected edges everywhere except the deadlock occurrence case for

bottom of the drawer. Only the vertexes corresponding to sides of the drawer have incoming edges, which mean that those need to be attached to the bottom of drawer before attaching them to any other parts.

The operations in decision sequences are examined against graph rules. The rules graph ensures that the decision sequences do not violate the physical constraints of the assembly sequence. In there is a deadlock violation error is displayed that is should be removed. Note that the rules graph is specific for assembly.

Appendix C

Source code

Algorithmic classes

Class HyperNode – HyperNode.cs

```
class HyperNode
    {
        private string name;
        private List<string> content;
        private List<NodePair> nodes;
        private int minWeight;
        private NodePair minPair;

        public HyperNode(string n, List<string> content)
        {
            this.name = n;
            this.content = content;
            this.nodes = new List<NodePair>();
            this.minWeight = 0;
            this.minPair = null;
        }
        // Empty Constructor
        public HyperNode()
        {
            this.name = "";
            this.content = new List<string>();
            this.nodes = new List<NodePair>();
            this.minWeight = 0;
            this.minPair = null;
        }

        public HyperNode(string n, List<string> content, List<NodePair> nodes)
        {
            this.name = n;
            this.content = content;
            this.nodes = nodes;
            this.minWeight = 0;
            this.minPair = null;
        }

        //copy constructor
        public HyperNode(HyperNode hypN)
        {
            this.name = hypN.name;
            this.content = hypN.content;
            this.nodes = hypN.nodes;
            this.minWeight = hypN.minWeight;
            this.minPair = null;
        }


        //Tree node to HyperNode constructor
        public HyperNode(BinaryTreeNode binN)
        {
            this.name = binN.getName();
            this.content = binN.getContent();
            this.nodes = new List<NodePair>();
            this.minWeight = binN.getWeight();
            this.minPair = null;
        }

        public string getRootName()
        {
            return this.name;
        }

        //Algorithm performs the comparision in O(n^2) time
        public bool checkNodesIdentical(HyperNode hypN)
        {
```

```
    if(this.content.Count != hypN.content.Count)
    {
        return false;
    }
    foreach (string tmp in this.content)
    {
        if (!hypN.content.Contains(tmp))
        {
            return false;
        }
    }
    return true;
}

//Algorithm performs the comparision in O(n^2) time
public bool checkNodesIdentical(BinaryTreeNode binN)
{
    if (this.content.Count != binN.getContent().Count)
    {
        return false;
    }
    foreach (string tmp in this.content)
    {
        if (!binN.getContent().Contains(tmp))
        {
            return false;
        }
    }
    return true;
}


public void buildHyperTree(List<BinaryTreeNode> binTrees)
{
    //assign the root name
    this.name = binTrees[0].getName();
    this.content = binTrees[0].getContent();

    //TraverseAndCopyTree(binTrees[0]);
    foreach(BinaryTreeNode cur_tree in binTrees)
    {
        this.processBinaryNode(cur_tree);
    }

    FindOptimumTimePath();
}


public void processBinaryNode(BinaryTreeNode binNode)
{
    HyperNode cur_node = findNode(this, binNode);

    HyperNode h_left = findNode(this, binNode.getLeft());
    HyperNode h_right = findNode(this, binNode.getRight());

    if (h_left == null && binNode.getLeft() != null)
    {
        h_left = new HyperNode(binNode.getLeft());
    }
    if (h_right == null && binNode.getRight() != null)
    {
        h_right = new HyperNode(binNode.getRight());
    }

    // If not a leaf
    if (h_left != null && h_right != null)
    {
        // Find if node Pair already exists
        bool b_create = true;
        foreach (NodePair nPair in cur_node.nodes)
        {
            if ((h_left == nPair.getLeft() && h_right == nPair.getRight()) ||
                (h_right == nPair.getLeft() && h_left == nPair.getRight()) )
            {
                b_create = false;
                break;
            }
```

```
            }
            if (b_create)
            {
                NodePair cur_pair = new NodePair(h_left, h_right,
                                                 binNode.getWeight());
                cur_node.nodes.Add(cur_pair);
            }
            // Go recursivly to left and right
            processBinaryNode(binNode.getLeft());
            processBinaryNode(binNode.getRight());
        }
    }

    bool isLeaf()
    {
        if (this.nodes.Count == 0)
        {
            return true;
        }
        if (this.nodes[0].getLeft() == null && this.nodes[0].getRight() == null)
        {
            return true;
        }
        return false;
    }

    public String outHyperTree_toList(HyperNode cur_node, int level, ListBox lb)
    {
        String outTab = new String(' ', level * 10);
        String outStr = "";
        foreach (NodePair nodeP in cur_node.nodes)
        {
            lb.Items.Add(outTab + nodeP.getLeft().name + " " + nodeP.getWeight());
            lb.Items.Add(outHyperTree_toList(nodeP.getLeft(), level + 1, lb));

            lb.Items.Add(outTab + nodeP.getRight().name + " " +
                                                 nodeP.getWeight());
            lb.Items.Add(outHyperTree_toList(nodeP.getRight(), level + 1, lb));

        }
        return outStr;
    }


    public HyperNode findNode(HyperNode cur_node, BinaryTreeNode binNode)
    {
        // If node doesnt exist
        if (binNode == null)
        {
            return null;
        }
        if (cur_node.checkNodesIdentical(binNode))
        {
            return cur_node;
        }

       HyperNode tmp_node;
       foreach(NodePair nodeP in cur_node.nodes)
       {
            if (nodeP.getLeft() != null)
            {
                tmp_node = findNode(nodeP.getLeft(), binNode);
                if (tmp_node != null) return tmp_node;
            }
            if (nodeP.getRight() != null)
            {
                tmp_node = findNode(nodeP.getRight(), binNode);
                if (tmp_node != null) return tmp_node;
            }
       }
       return null;
    }
```

```
    //implementation of the optimal time and path finding

     public String printMinPath(HyperNode cur_node, int level)
     {
         String outTab = new String(' ', level * 10);
         String outStr = "";
         if(cur_node.minPair != null)
         {
             NodePair nodeP = cur_node.minPair;
             outStr += outTab + nodeP.getLeft().name + " " +
                               nodeP.getWeight() + Environment.NewLine;
             outStr += printMinPath(nodeP.getLeft(), level + 1);

             outStr += outTab + nodeP.getRight().name + " " +
                                nodeP.getWeight() + Environment.NewLine;
             outStr += printMinPath(nodeP.getRight(), level + 1);

         }
         return outStr;
     }


     void FindOptimumTimePath()
     {

         MessageBox.Show(FindOptimumTimePathRecurrent(this).ToString());
         MessageBox.Show(printMinPath(this, 0));
         Clipboard.SetText(printMinPath(this, 0));
     }

     //Impementing the recurrent expression
     int FindOptimumTimePathRecurrent( HyperNode currNode)
     {
         int value = int.MaxValue;
         int curNodeP_val;
         NodePair minPair = null;

         if (currNode.isLeaf())
         {
             return 0;
         }
         if (currNode.minPair != null)
         {
             return currNode.minWeight;
         }

         foreach(NodePair curNodeP in currNode.nodes)
         {
             curNodeP_val = 0;
             if (curNodeP.getLeft() != null && curNodeP.getRight() != null)
             {
                 curNodeP_val += curNodeP.getWeight()
                     + FindOptimumTimePathRecurrent(curNodeP.getLeft())
                     + FindOptimumTimePathRecurrent(curNodeP.getRight());
             }
             if (curNodeP_val < value)
             {
                 minPair = curNodeP;
                 value = curNodeP_val;
             }
         }

         currNode.minWeight = value;
         currNode.minPair = minPair;

         return value;
     }
}
```

Class NodePair – NodePair.cs

```
  class NodePair
    {
        private int weight;
        private HyperNode left;
        private HyperNode right;

        public NodePair(HyperNode l, HyperNode r, int w)
        {
            this.left = l;
            this.right = r;
            this.weight = w;
        }

        public int getWeight() { return this.weight; }
        public void setWeight(int w) { this.weight = w; }

        public HyperNode getLeft() { return this.left; }
        public void assignLeft(HyperNode l) { this.left = new HyperNode(l); }

        public HyperNode getRight() { return this.right; }
        public void assingRight(HyperNode r) { this.right = new HyperNode(r); }

    }
```

Class NaryTreeNode – NaryTreeNode.cs

```
public class NaryTreeNode
    {
        string name;
        List<string> content;
        List<NaryTreeNode> children;
        NaryTreeNode parent;
        //the weight from parent to oneself
        int weight;

        public NaryTreeNode(string n, NaryTreeNode p, List<string> c)
        {
            this.name = n;
            this.parent = p;
            this.weight = 0;
            this.content = c;
            this.children = new List<NaryTreeNode>();
        }

        public NaryTreeNode(string n, NaryTreeNode p, int w, List<string> c)
        {
            this.name = n;
            this.parent = p;
            this.weight = w;
            this.content = c;
            this.children = new List<NaryTreeNode>();
        }

        public string getName() { return name; }
        public void setName(string str) { this.name = str; }
        public NaryTreeNode getParent() { return parent; }
        public void setParent(NaryTreeNode p) { this.parent = p; }

        public List<string> getContent() { return this.content; }
        public void setContent(List<string> c) { this.content = c; }

        public List<NaryTreeNode> getChildren() { return this.children;}
        public void setChildren(List<NaryTreeNode> ch){this.children = ch;}


    }
```

Class NaryTree – NaryTree.cs

```
public class NaryTree
    {
        List<string> leafs;
        NaryTreeNode Root;

        public NaryTree(NaryTreeNode root, List<string> leafs)
        {
            this.Root = root;
            this.leafs = leafs;
        }

        public NaryTreeNode searchNamePreorder(string name, NaryTreeNode parent)
        {
            if (parent.getName() == name)
            {
                return parent;
            }

            foreach (NaryTreeNode tr in parent.getChildren())
            {
                return searchNamePreorder(name, tr);
            }
            //never reaches this point
            return null;
        }

    }
```

Class BinaryTreeNode – BinaryTreeNode.cs

```
    public class BinaryTreeNode
    {
        string name;
        List<string> content;
        //TreeNode parent;
        BinaryTreeNode left;
        BinaryTreeNode right;

        //the weight from parent to oneself
        int weight;
        int sum_weight;


        public BinaryTreeNode()
        {
            this.name = "";
            //this.parent = p;
            this.weight = 0;
            this.sum_weight = 0;
            this.content = new List<string>();
        }

        public BinaryTreeNode(string n, List<string> c)
        {
            this.name = n;
            //this.parent = p;
            this.weight = 0;
            this.sum_weight = 0;
            this.content = c;
            this.left = null;
            this.right = null;
        }


        public BinaryTreeNode(string n, int w, List<string> c)
        {
            this.name = n;
            //this.parent = p;
            this.weight = w;
            this.sum_weight = 0;
            this.content = new List<string>(c);
            this.left = null;
            this.right = null;
        }

        //copy constructor
        public BinaryTreeNode(BinaryTreeNode BTN)
        {
            this.name = BTN.name;
            //this.parent = p;
            this.weight = BTN.weight;
            this.sum_weight = BTN.sum_weight;
            this.content = BTN.content;
            this.left = BTN.left;
            this.right = BTN.right;
        }

        public int getWeight() { return this.weight; }
        public void setWeight(int w) { this.weight = w; }

        public string getName() { return name; }
        public void setName(string str) { this.name = str; }
        //public BinaryTreeNode getParent() { return parent; }
        //public void setParent(BinaryTreeNode p) { this.parent = p; }

        public List<string> getContent() { return this.content; }
        public void setContent(List<string> c) { this.content = c; }

        public BinaryTreeNode getLeft() { return this.left; }
        public void assignLeft(BinaryTreeNode l) { this.left = new BinaryTreeNode(l);
}

        public BinaryTreeNode getRight() { return this.right; }
        public void assingRight(BinaryTreeNode r) {
                            this.right = new BinaryTreeNode(r); }
```

```
    public BinaryTreeNode searchNamePreorder(string name, BinaryTreeNode parent)
    {
        if (parent.getName() == name)
        {
            return parent;
        }

        BinaryTreeNode tmp_node;
        if (parent.getLeft() != null)
        {
            tmp_node = searchNamePreorder(name, parent.getLeft());
            if (tmp_node != null) return tmp_node;
        }
        if (parent.getRight() != null)
        {
            tmp_node = searchNamePreorder(name, parent.getRight());
            if (tmp_node != null) return tmp_node;
        }
        return null;
    }

    public String outBinaryTree(BinaryTreeNode cur_node, int level)
    {
        String outTab = new String(' ', level * 10);
        String outStr = "";
        BinaryTreeNode temp;

        if (cur_node.getLeft() != null)
        {
            temp = cur_node.getLeft();
            string forstr ="";

            foreach(string str in temp.content)
            { forstr += str + " ";}
            outStr += outTab + temp.name + " " + temp.weight + " SW:" +
                      temp.sum_weight + " "+ forstr + Environment.NewLine;
            outStr += outBinaryTree(temp, level + 1);
        }
        if (cur_node.getRight() != null)
        {
            temp = cur_node.getRight();
            string forstr ="";

            foreach(string str in temp.content)
            { forstr += str + " ";}
            outStr += outTab + temp.name + " " + temp.weight + " SW:" +
                        temp.sum_weight + " " + forstr + Environment.NewLine;
            outStr += outBinaryTree(temp, level + 1);
        }
        return outStr;
    }


    public void setSumWeights()
    {
        //For root
        this.sum_weight = this.weight;
        setSumWeightsRecursive(this);
    }

    private void setSumWeightsRecursive(BinaryTreeNode curNode)
    {
        if (curNode.left != null)
        {
            curNode.left.sum_weight = curNode.sum_weight + curNode.left.weight;
            setSumWeightsRecursive(curNode.left);
        }
        if (curNode.right != null)
        {
            curNode.right.sum_weight = curNode.sum_weight + curNode.right.weight;
            setSumWeightsRecursive(curNode.right);
        }
    }

}
```

Main visualization classes

Class Line – Line.cs

```
    public class Line
    {
        Point start;
        Point end;
        const double minVal = 0.1;
        const int calcOffset = 4;

        public Line()
        {
            this.start = new Point(0, 0);
            this.end = new Point(0, 0);
        }

        public Line(Point st, Point e)
        {
            this.start = st;
            this.end = e;
        }

        int compute_a()
        {
            return (int)((end.Y - start.Y) / (end.X - start.X + minVal));
        }

        int compute_b()
        {
            return (int)((end.X * start.Y - start.X * end.Y) /
                            (end.X - start.X + minVal));
        }

        public Point TwoLinesIntersect(Line l2)
        {
            double a = (this.end.Y - this.start.Y) /
                       (this.end.X - this.start.X + minVal);
            double b = (this.end.X * this.start.Y - this.start.X * this.end.Y) /
                       (this.end.X - this.start.X + minVal);
            double c = (l2.end.Y - l2.start.Y) / (l2.end.X - l2.start.X + minVal);
            double d = (l2.end.X * l2.start.Y - l2.start.X * l2.end.Y) /
                        (l2.end.X - l2.start.X + minVal);
            Point p;
            //P1.y = a * P1.x + b
            if ((Math.Abs(a - c) < 0.2) ||
               ((this.start.X == this.end.X) && (l2.start.X == l2.end.X)))
            {
                p = new Point(-1, -1);
            }
            else
            {
                int x = (int)((d - b) / (a - c + minVal));
                int y = (int)((a * d - b * c) / (a - c + minVal));
                p = new Point(x, y);
            }
            return p;
        }

        void reverseLineIfNecessary(ref Line l)
        {
            if (l.start.X > l.end.X)
            {
                Point temp = l.end;
                l.end = l.start;
                l.start = temp;
            }
            else if (l.start.X == l.end.X)
            {
                if (l.start.Y > l.end.Y)
                {
                    Point temp = l.end;
                    l.end = l.start;
                    l.start = temp;
                }
            }
        }
```

```
void reversePointsIfNecessary(ref Point p1, ref Point p2)
{
    if (p1.X > p2.X)
    {
        Point temp = p2;
        p2 = p1;
        p1 = temp;
    }
    else if (p1.X == p2.X)
    {
        if (p1.Y > p2.Y)
        {
            Point temp = p2;
            p2 = p1;
            p1 = temp;
        }
    }
}

bool isolateSelectionFromRest(Line l_dr, Line l_temp, ref Line toCut,
                                                        ref int n_count)
{
    if (isIntersectionLine(l_temp, l_dr))
    {
        //sort the line incrementaly
        reverseLineIfNecessary(ref l_temp);
        reverseLineIfNecessary(ref l_dr);
        Point s1 = new Point(Math.Max(l_temp.start.X, l_dr.start.X),
                             Math.Max(l_temp.start.Y, l_dr.start.Y));
        Point s2 = new Point(Math.Min(l_temp.end.X, l_dr.end.X),
                             Math.Min(l_temp.end.Y, l_dr.end.Y));

        if (Math.Abs(s1.X - s2.X) < calcOffset &&
            Math.Abs(s1.Y - s2.Y) < calcOffset)
        {
            return false;
        }

        //includes the whole line
        if (Math.Abs(l_dr.start.X - s1.X) < calcOffset &&
            Math.Abs(l_dr.start.Y - s1.Y) < calcOffset &&
            Math.Abs(l_dr.end.X - s2.X) < calcOffset &&
            Math.Abs(l_dr.end.Y - s2.Y) < calcOffset)
        {
            toCut.start = toCut.end;
            n_count++;
            return true;
        }

        else if (Math.Abs(l_dr.start.X - s1.X) < calcOffset &&
                 Math.Abs(l_dr.start.Y - s1.Y) < calcOffset)
        {
            toCut.start = s2;
            n_count++;
            return true;
        }
        else if (Math.Abs(l_dr.end.X - s2.X) < calcOffset &&
           Math.Abs(l_dr.end.Y - s2.Y) < calcOffset)
        {
            toCut.end = s1;
            n_count++;
            return true;
        }
    }
    return false;
}

bool isFirstPointGreater(Point p1, Point p2)
{
    if (p1.X > p2.X)
    {
        return true;
    }
    else if (p1.X < p2.X)
    {
        return false;
```

```
    }
    else
    {
        return (p1.Y > p2.Y);
    }
}

public void resortNeigbhborsMinXY(ref Drawing[] neighbors)
{
    Drawing temp;

    for (int i = neighbors.Length; i > 0; i--)
    {
        for (int j = 1; j < i; j++)
        {
            //Point comparision next
            if (neighbors[j - 1] != null && neighbors[j] != null) {
                Point p1 = neighbors[j-1].getMinPossibleCoordinte();
                Point p2 = neighbors[j].getMinPossibleCoordinte();
                if( isFirstPointGreater(p1, p2))
                {
                      temp = neighbors[j-1];
                      neighbors[j-1] = neighbors[j];
                      neighbors[j] = temp;
                }
            }
        }
    }
}


public bool isDeadlockPresent(Drawing dr, Drawing[] neighbors)
{
    //the arrays initialized for
    Point[] tempPoints;
    Line[] tempLines;
    int n_count = 0;

    Point[] drPoints = dr.getContPoints();
    Line[] drSides = new Line[drPoints.Length];
    for (int i = 0; i < drPoints.Length; i++)
    {
        drSides[i] = new Line(drPoints[i], drPoints[(i + 1) %
                                    drPoints.Length]);
    }


    //Do the extraction
    int tm = 0;
    int si = 0;
    Line l_temp;
    Line l_dr;

    //sort the neighbors ascending by their (x,y) coordinate
    resortNeigbhborsMinXY(ref neighbors);


    while (si < drSides.Length)
    {
        l_dr = drSides[si];

        for (int i = 0; i < neighbors.Length; i++)
        {
            tm = 0;
            Drawing tempDr = neighbors[i];
            tempPoints = tempDr.getContPoints();
            tempLines = new Line[tempPoints.Length];
            for (int j = 0; j < tempPoints.Length; j++)
            {
                tempLines[j] = new Line(tempPoints[j],
                                tempPoints[(j + 1) % tempPoints.Length]);
            }

            while (tm < tempLines.Length)
            {
                l_temp = tempLines[tm];
                //Assign start to end if the line coindsides
```

```
                if (isolateSelectionFromRest(l_dr, l_temp, ref drSides[si],
                                             ref n_count))
                {
                    //neighbors.Remove(tempDr);
                    break;
                }
                tm++;
            }
        }
        si++;
    }

    foreach (Line l in drSides)
    {
        if (l.start != l.end)
        {
            return false;
        }
    }
    return true;
}


//short 0- no overlap, 1- line overlap, 2- point overlap,
public short LinesArraysIntersect(Line[] arr1, Line[] arr2)
{
    Point pt;
    for (int i = 0; i < arr1.Length; i++)
    {
        for (int j = 0; j < arr2.Length; j++)
        {
            pt = arr1[i].TwoLinesIntersect(arr2[j]);
            //                    pt = arr2[j].TwoLinesIntersect(arr1[i]);
            if (pt.X == 0 || pt.Y == 0)
            {
                if (isIntersectionLine(arr1[i], arr2[j]))
                {
                    return 1;
                }
            }
            if (arr1[i].isPointOnLine(pt) && arr2[j].isPointOnLine(pt))
            {
                return 2;
            }
        }
    }
    return 0;
}


//short 0- no overlap, 1- line overlap, 2- point overlap,
public short pointsArraysIntersect(Point[] arr1, Point[] arr2)
{
    Point pt;
    Line l1;
    Line l2;
    Point par = new Point(-1, -1);
    short valIntersect = 0;

    for (int i = 0; i < arr1.Length; i++)
    {
        l1 = new Line(arr1[i], arr1[(i + 1) % arr1.Length]);

        for (int j = 0; j < arr2.Length; j++)
        {
            l2 = new Line(arr2[j], arr2[(j + 1) % arr2.Length]);

            pt = l1.TwoLinesIntersect(l2);

            //case parralel
            if (pt.X == 0 || pt.Y == 0 || (pt.X == -1 && pt.Y == -1))
            {
                if (isIntersectionLine(l1, l2))
                {
                    valIntersect = 1;
                    return valIntersect;
                }
```

```
                }
                if (l1.isPointOnLine(pt) && l2.isPointOnLine(pt))
                {
                    valIntersect = 2;
                }
            }
        }
        return valIntersect;
    }

    public bool isIntersectionLine(Line l1, Line l2)
    {
        return (l1.isPointOnLine(l2.start) || l1.isPointOnLine(l2.end) ||
                l2.isPointOnLine(l1.start) || l2.isPointOnLine(l1.end));
    }


    public bool isPointOnLine(Point p)
    {
        if ((p.X >= Math.Min(this.start.X, this.end.X) - calcOffset) &&
            (p.X <= Math.Max(this.start.X, this.end.X) + calcOffset) &&
            (p.Y >= Math.Min(this.start.Y, this.end.Y) - calcOffset) &&
            (p.Y <= Math.Max(this.start.Y, this.end.Y) + calcOffset))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

Class Drawing – Drawing.cs

```
    public class Drawing
    {
        private int id;
        SolidBrush fillBrush;
        Pen linePen;
        Point[] contPoints;
        Point center;
        Dictionary<int, double> neighbors;
        string name;

        //Default
        const int offset = 10;

        //for file load
        public Drawing(int count)
        {
            this.id = 0;
            this.fillBrush = new SolidBrush(Color.White);
            this.linePen = new Pen(Color.Black, 4);
            contPoints = new Point[count];
            center = new Point(0, 0);
            neighbors = new Dictionary<int, double>();
            name = "";
        }

        public Drawing(int identificator, int count)
        {
            this.id = identificator;
            this.fillBrush = new SolidBrush(Color.White);
            this.linePen = new Pen(Color.Black, 4);
            contPoints = new Point[count];
            center = new Point(0, 0);
            neighbors = new Dictionary<int, double>();
            name = "";
        }

        public Drawing(Point start, Point end, int count)
        {
            this.id = 0;
            this.fillBrush = new SolidBrush(Color.White);
            this.linePen = new Pen(Color.Black, 4);
            contPoints = new Point[count];
            center = new Point(0, 0);
            neighbors = new Dictionary<int, double>();
            name = "";
        }

         //copy
        public Drawing(Drawing draw)
        {
            this.id = draw.id;
            this.fillBrush = draw.fillBrush;
            this.linePen = draw.linePen;
            this.contPoints = (Point[])draw.contPoints.Clone();
            this.center = draw.center;
            this.neighbors = draw.neighbors;
            name = "";
        }

        public Drawing(Color fill, int ident, int count)
        {
            this.id = ident;
            this.fillBrush = new SolidBrush(fill);
            this.linePen = new Pen(Color.Black, 4);
            contPoints = new Point[count];
            center = new Point(0, 0);
            neighbors = new Dictionary<int, double>();
            name = "";
        }

        public Drawing(Color fill, Color line, int count)
        {
            this.id = 0;
            this.fillBrush = new SolidBrush(fill);
            this.linePen = new Pen(line, 4);
```

```
    contPoints = new Point[count];
    center = new Point(0, 0);
    neighbors = new Dictionary<int, double>();
    name = "";
}

public void setId(int _id)
{
    this.id = _id;
}

public int getId()
{
    return this.id;
}

public void setFillColor(Color col)
{
    this.fillBrush = new SolidBrush(col);
}

public Color getFillColor()
{
    return this.fillBrush.Color;
}

public void setlinePen(Color col)
{
    this.linePen = new Pen(col, 4);
}

public Pen getlinePen()
{
    return this.linePen;
}

public void setContPoints(Point[] pts)
{
    this.contPoints = pts;
}

public Point[] getContPoints()
{
    return this.contPoints;
}

public void setCenter(Point pt)
{
    this.center = pt;
}

public Point getCenter()
{
    return this.center;
}

public void setName(string st)
{
    this.name = st;
}

public string getName()
{
    return this.name;
}

public void setNeighbors(List<int> bors)
{
    foreach (int i in bors)
    {
        this.neighbors.Add(i, 0);
    }

}

public Dictionary<int, double> getNeighbors()
{
```

```
    return this.neighbors;
}

 public short checkDrawingOverlap(List<Drawing> drList)
{
    int thisArrLen = this.contPoints.Length;
    Line l = new Line();
    short intersectVal = 0;
    this.neighbors.Clear();
    foreach (Drawing dr in drList)
    {
        intersectVal = l.pointsArraysIntersect(dr.contPoints,
                                this.contPoints);
        dr.neighbors.Remove(this.id);
        //if intersected by point not line return immidiately
        if(intersectVal == 2)
        {
            return intersectVal;
        }
        else if(intersectVal == 1)
        {
            this.neighbors.Add(dr.id, 0);
            dr.neighbors.Add(this.id, 0);
        }
    }
    return 0;
}

public Drawing getOutherDrawing()
{
    Drawing dr = new Drawing(this);
    for (int i = 0; i < dr.contPoints.Length; i++)
    {
        if (dr.contPoints[i].X < dr.center.X)
        {
            dr.contPoints[i].X -= offset;
        }
        else
        {
            dr.contPoints[i].X += offset;
        }
        if (dr.contPoints[i].Y < dr.center.Y)
        {
            dr.contPoints[i].Y -= offset;
        }
        else
        {
            dr.contPoints[i].Y += offset;
        }
    }
    dr.updateCenter();
    return dr;
}

public void DrawObjectName(Graphics gr)
{
    Font nFont = new Font("Arial", 14, GraphicsUnit.Pixel);
    Brush nBrush = new SolidBrush(Color.Black);
    double cnt = name.Length;
    int strWidth = (int)gr.MeasureString(name, nFont).Width;
    int strHight = (int)gr.MeasureString(name, nFont).Height;
    gr.DrawString(name, nFont, nBrush,
                new Point(center.X - strWidth / 2, center.Y - strHight / 2));
}

public Point getMinPossibleCoordinte()
{
    int min_x = int.MaxValue;
    int min_y = int.MaxValue;
    //case error occured
    if (this.contPoints.Length < 3 || this.contPoints.Length > 4)
    {
        return new Point(-1, -1);
    }

    for(int i = 0; i < this.contPoints.Length; i++)
    {
```

```
            if(this.contPoints[i].X < min_x)
            {
                min_x = this.contPoints[i].X;
            }
            if(this.contPoints[i].Y < min_y)
            {
                min_y = this.contPoints[i].Y;
            }
        }
        return new Point(min_x, min_y);
    }


    public void DrawShowMessage(Graphics gr, string msg)
    {
        Font nFont = new Font("Arial", 14, GraphicsUnit.Pixel);
        Brush nBrush = new SolidBrush(Color.Red);
        double cnt = name.Length;
        int strWidth = (int)gr.MeasureString(msg, nFont).Width;
        int strHight = (int)gr.MeasureString(msg, nFont).Height;
        gr.DrawString(name, nFont, nBrush,
                    new Point(center.X - strWidth / 2, center.Y - strHight / 2));
    }

    public virtual void updateCenter() { }
    public virtual void Draw(Graphics gr)
    {
        gr.DrawPolygon(this.linePen, this.contPoints);
        gr.FillPolygon(this.fillBrush, this.contPoints);
    }

    public virtual bool isPointInside(Point pnt) {
        Point min = new Point(10000, 10000);
        Point max = new Point(0, 0);
        foreach (Point pt in this.contPoints)
        {
            if (pt.X < min.X)
                min.X = pt.X;
            if (pt.Y < min.Y)
                min.Y = pt.Y;
            if (pt.X > max.X)
                max.X = pt.X;
            if (pt.Y > min.Y)
                max.Y = pt.Y;
        }
        return ((pnt.X > min.X) && (pnt.X < max.X) &&
                  (pnt.Y > min.Y) && (pnt.Y < max.Y));
    }
    public virtual void rotate() { }
    public virtual void move() { }
    public virtual void resizeUpdate() { }
    public virtual int ofType()
    {
        return 0; //1 rectangle, 2 triangle, 3 rhumbus
    }


    private XmlAttribute CreateAttribute(XmlDocument doc, string aname,
                                         string avalue)
    {
        XmlAttribute t_attrib = doc.CreateAttribute(aname);
        t_attrib.Value = avalue;
        return t_attrib;
    }

    public virtual void WriteXML(XmlDocument doc, XmlElement elem)
    {

        elem.Attributes.Append(CreateAttribute(doc, "id", this.id.ToString()));
        elem.Attributes.Append(CreateAttribute(doc, "name", this.name));

        // adding fill color
        XmlElement cur_elem = doc.CreateElement("fill_color");

        cur_elem.Attributes.Append(CreateAttribute(doc, "r",
                    this.fillBrush.Color.R.ToString()));
        cur_elem.Attributes.Append(CreateAttribute(doc, "g",
```

```
                this.fillBrush.Color.G.ToString()));
        cur_elem.Attributes.Append(CreateAttribute(doc, "b",
                this.fillBrush.Color.B.ToString()));
        cur_elem.Attributes.Append(CreateAttribute(doc, "a",
                this.fillBrush.Color.A.ToString()));

        elem.AppendChild(cur_elem);

        // adding line pen
        cur_elem = doc.CreateElement("line_pen");

        cur_elem.Attributes.Append(CreateAttribute(doc, "r",
                this.linePen.Color.R.ToString()));
        cur_elem.Attributes.Append(CreateAttribute(doc, "g",
                this.linePen.Color.G.ToString()));
        cur_elem.Attributes.Append(CreateAttribute(doc, "b",
                this.linePen.Color.B.ToString()));
           cur_elem.Attributes.Append(CreateAttribute(doc, "width",
                this.linePen.Width.ToString()));

        elem.AppendChild(cur_elem);

        // adding center
        cur_elem = doc.CreateElement("center");

        cur_elem.Attributes.Append(CreateAttribute(doc, "x",
                this.center.X.ToString()));
        cur_elem.Attributes.Append(CreateAttribute(doc, "y",
                this.center.Y.ToString()));

        elem.AppendChild(cur_elem);

        // adding points
        cur_elem = doc.CreateElement("point_list");

        foreach(Point p in this.contPoints)
        {
            XmlElement point_element = doc.CreateElement("point");
            point_element.Attributes.Append(CreateAttribute(doc, "x",
                   p.X.ToString()));
            point_element.Attributes.Append(CreateAttribute(doc, "y",
                   p.Y.ToString()));
            cur_elem.AppendChild(point_element);
        }

        elem.AppendChild(cur_elem);

        // adding neighbors
        cur_elem = doc.CreateElement("neighbors");

        foreach(int i in this.neighbors.Keys) //(int i in this.neighbors)
        {
            XmlElement neighbors_element = doc.CreateElement("ref");
            neighbors_element.InnerText = i.ToString();
            cur_elem.AppendChild(neighbors_element);
        }

        elem.AppendChild(cur_elem);
    }


     public void LoadXML(XmlElement el)
     {
        //load id
        this.id = Int32.Parse(el.Attributes["id"].Value);

        //load name
        this.name = el.Attributes["name"].Value.ToString();

        //load fill brush
        this.fillBrush.Color = Color.FromArgb(Int32.Parse(
                   el.GetElementsByTagName("fill_color")[0].Attributes["a"].Value),
           Int32.Parse(
                   el.GetElementsByTagName("fill_color")[0].Attributes["r"].Value),
           Int32.Parse(
                   el.GetElementsByTagName("fill_color")[0].Attributes["g"].Value),
           Int32.Parse(
```

```
            el.GetElementsByTagName("fill_color")[0].Attributes["b"].Value));


         //load line pen
            this.linePen.Color = Color.FromArgb(
            Int32.Parse(
                   el.GetElementsByTagName("line_pen")[0].Attributes["r"].Value),
            Int32.Parse(
                   el.GetElementsByTagName("line_pen")[0].Attributes["g"].Value),
            Int32.Parse(
                   el.GetElementsByTagName("line_pen")[0].Attributes["b"].Value));

            this.linePen.Width =
            Int32.Parse(el.GetElementsByTagName("line_pen")[0].
                   Attributes["width"].Value);


         //load neighbors
         foreach (XmlElement pt_el in el.GetElementsByTagName(
                                                 "neighbors")[0].ChildNodes)
         {
             this.neighbors.Add(Int32.Parse(pt_el.InnerText), 0);
         }
      }
   }
```

Class Rectangle – Rectangle.cs

```
    public class Rectangle : Drawing
    {
        Point[] pts;
        private Point rcCenter;


        public Rectangle(XmlElement el) : base(4)
        {

            rcCenter = new Point(Int32.Parse(
               el.GetElementsByTagName("center")[0].Attributes["x"].Value),
                              Int32.Parse(
               el.GetElementsByTagName("center")[0].Attributes["y"].Value));

            //load points
            pts = new Point[4];
            int i = 0;
            foreach (XmlElement pt_el in el.GetElementsByTagName("point_list")[0].
                                                                        ChildNodes)
            {
                pts[i] = new Point(Int32.Parse(pt_el.Attributes["x"].Value),
                                   Int32.Parse(pt_el.Attributes["y"].Value));
                i++;
            }
            base.setContPoints(pts);
            base.setCenter(this.rcCenter);
        }

        public Rectangle(Point[] cPoints, int id) : base(id, 4)
        {
            this.pts = (Point[])cPoints.Clone();
            this.rcCenter = new Point((pts[0].X + pts[2].X) / 2,
                                      (pts[0].Y + pts[2].Y) / 2);
            base.setContPoints(pts);
            base.setCenter(this.rcCenter);
        }


        public Rectangle(Point tl, Point br, int id) : base(id, 4)
        {
            pts = new Point[4];

            pts[0] = new Point(Math.Min(tl.X, br.X), Math.Min(tl.Y, br.Y));
            pts[2] = new Point(Math.Max(tl.X, br.X), Math.Max(tl.Y, br.Y));
            pts[1] = new Point(pts[2].X, pts[0].Y);
            pts[3] = new Point(pts[0].X, pts[2].Y);

            this.rcCenter = new Point((pts[0].X + pts[2].X) / 2,
                                      (pts[0].Y + pts[2].Y) / 2);
            base.setContPoints(pts);
            base.setCenter(this.rcCenter);
        }

        public Rectangle(Point tl, Point br, Color fill, int id)
            : base(fill, id, 4)
        {
            pts = new Point[4];
            pts[0] = new Point(Math.Min(tl.X, br.X), Math.Min(tl.Y, br.Y));
            pts[2] = new Point(Math.Max(tl.X, br.X), Math.Max(tl.Y, br.Y));
            pts[1] = new Point(pts[2].X, pts[0].Y);
            pts[3] = new Point(pts[0].X, pts[2].Y);
            this.rcCenter = new Point((pts[0].X + pts[2].X) / 2,
                                      (pts[0].Y + pts[2].Y) / 2);

            base.setContPoints(pts);
            base.setCenter(this.rcCenter);
        }

        public override void resizeUpdate()
        {

            pts[1] = new Point(pts[2].X, pts[0].Y);
            pts[3] = new Point(pts[0].X, pts[2].Y);
            updateCenter();
```

```
            base.setContPoints(pts);
            base.setCenter(rcCenter);
        }

        public override void updateCenter()
        {
            this.rcCenter = new Point((pts[0].X + pts[2].X) / 2,
                                      (pts[0].Y + pts[2].Y) / 2);
        }


        public override void Draw(Graphics gr)
        {
            gr.DrawRectangle(getlinePen(), pts[0].X, pts[0].Y, pts[2].X –
                                    pts[0].X, pts[2].Y - pts[0].Y);
            gr.FillRectangle(new SolidBrush(getFillColor()), pts[0].X, pts[0].Y,
                                     pts[2].X - pts[0].X, pts[2].Y - pts[0].Y);
            base.DrawObjectName(gr);
        }

        public override bool isPointInside(Point pnt)
        {
            return (pnt.X > Math.Min(pts[0].X, pts[2].X) &&
                             pnt.X < Math.Max(pts[0].X, pts[2].X) &&
                    pnt.Y > Math.Min(pts[1].Y, pts[3].Y) &&
                             pnt.Y < Math.Max(pts[1].Y, pts[3].Y));
         }

        public override void rotate()
        {
            int min_coord = Math.Min((pts[2].X - pts[0].X), (pts[2].Y - pts[0].Y));
            int max_coord = Math.Max((pts[2].X - pts[0].X), (pts[2].Y - pts[0].Y));
            if (pts[2].X - pts[0].X > pts[2].Y - pts[0].Y)
            {
                pts[0] = new Point(pts[0].X + (max_coord - min_coord) / 2,
                                   pts[0].Y - (max_coord - min_coord) / 2);
                pts[2] = new Point(pts[2].X - (max_coord - min_coord) / 2,
                                   pts[2].Y + (max_coord - min_coord) / 2);
                pts[1] = new Point(pts[2].X, pts[0].Y);
                pts[3] = new Point(pts[0].X, pts[2].Y);

            }
            else
            {
                pts[0] = new Point(pts[0].X - (max_coord - min_coord) / 2,
                                    pts[0].Y + (max_coord - min_coord) / 2);
                pts[2] = new Point(pts[2].X + (max_coord - min_coord) / 2,
                                   pts[2].Y - (max_coord - min_coord) / 2);
                pts[1] = new Point(pts[2].X, pts[0].Y);
                pts[3] = new Point(pts[0].X, pts[2].Y);
            }
            this.rcCenter = new Point((pts[0].X + pts[2].X) / 2,
                                      (pts[0].Y + pts[2].Y) / 2);

            //update base
            base.setContPoints(pts);
            base.setCenter(this.rcCenter);
        }

        public override void move()
        {
            this.pts = base.getContPoints();
            this.rcCenter = base.getCenter();
        }

        public override int ofType()
        {  return 1; }

        public override void WriteXML(XmlDocument doc, XmlElement elem)
        {
            elem = doc.CreateElement("rectangle");
            base.WriteXML(doc, elem);
            doc.DocumentElement.AppendChild(elem);
        }
    }
}
```

Class Rhombus – Rhombus.cs

```
   public class Rhombus : Drawing
    {
        Point[] rmLocs;
        Point rmCenter;


        public Rhombus(XmlElement el) : base(4)
        {

            rmCenter = new Point(Int32.Parse(el.
               GetElementsByTagName("center")[0].Attributes["x"].Value),
               Int32.Parse(el.GetElementsByTagName("center")[0].Attributes["y"].Value)
               );

            //load points
            rmLocs = new Point[4];
            int i = 0;
            foreach (XmlElement pt_el in el.
               GetElementsByTagName("point_list")[0].ChildNodes)
            {
                rmLocs[i] = new Point(Int32.Parse(pt_el.Attributes["x"].Value),
                                Int32.Parse(pt_el.Attributes["y"].Value));
                i++;
            }
            base.setContPoints(rmLocs);
            base.setCenter(rmCenter);
        }

        public Rhombus(Point[] cPoints, int id) : base(id, 4)
        {
            this.rmLocs = (Point[])cPoints.Clone();
            this.rmCenter = new Point((rmLocs[0].X + rmLocs[2].X) / 2,
                                      (rmLocs[1].Y + rmLocs[3].Y) / 2);
            base.setContPoints(rmLocs);
            base.setCenter(rmCenter);
        }


        public Rhombus(Point s, Point e, int id) : base(id, 4)
        {
            rmLocs = new Point[4];
            int min_X = Math.Min(s.X, e.X);
            int min_Y = Math.Min(s.Y, e.Y);
            int max_X = Math.Max(s.X, e.X);
            int max_Y = Math.Max(s.Y, e.Y);

            this.rmLocs[0] = new Point(min_X, (min_Y + max_Y) / 2);
            this.rmLocs[1] = new Point((min_X + max_X) / 2, max_Y);
            this.rmLocs[2] = new Point(max_X, (min_Y + max_Y) / 2);
            this.rmLocs[3] = new Point((min_X + max_X) / 2, min_Y);
            this.rmCenter = new Point((rmLocs[0].X + rmLocs[2].X) / 2,
                                      (rmLocs[1].Y + rmLocs[3].Y) / 2);

            base.setContPoints(rmLocs);
            base.setCenter(this.rmCenter);
        }

        public Rhombus(Point s, Point e, Color fill, int id)
            : base(fill, id, 4)
        {
            rmLocs = new Point[4];
            int min_X = Math.Min(s.X, e.X);
            int min_Y = Math.Min(s.Y, e.Y);
            int max_X = Math.Max(s.X, e.X);
            int max_Y = Math.Max(s.Y, e.Y);

            this.rmLocs[0] = new Point(min_X, (min_Y + max_Y) / 2);
            this.rmLocs[1] = new Point((min_X + max_X) / 2, max_Y);
            this.rmLocs[2] = new Point(max_X, (min_Y + max_Y) / 2);
            this.rmLocs[3] = new Point((min_X + max_X) / 2, min_Y);
            this.rmCenter = new Point((min_X + max_X) / 2, (min_Y + max_Y) / 2);

            base.setContPoints(rmLocs);
            base.setCenter(this.rmCenter);
        }
```

```
void updateAllPoints(int min_X, int min_Y, int max_X, int max_Y)
{
    for (int i = 0; i < 4; i++)
    {
        if (rmLocs[i].X <= min_X) { min_X = rmLocs[i].X; }
        if (rmLocs[i].X >= max_X) { max_X = rmLocs[i].X; }
        if (rmLocs[i].Y <= min_Y) { min_Y = rmLocs[i].Y; }
        if (rmLocs[i].Y >= max_Y) { max_Y = rmLocs[i].Y; }
    }

    this.rmLocs[0] = new Point(min_X, (min_Y + max_Y) / 2);
    this.rmLocs[1] = new Point((min_X + max_X) / 2, max_Y);
    this.rmLocs[2] = new Point(max_X, (min_Y + max_Y) / 2);
    this.rmLocs[3] = new Point((min_X + max_X) / 2, min_Y);
    this.rmCenter = new Point((min_X + max_X) / 2, (min_Y + max_Y) / 2);
}


int rhombusArea(Point p1, Point p2, Point p3, Point p4)
{
    double d1 = Math.Sqrt((p1.X - p3.X) * (p1.X - p3.X) +
                          (p1.Y - p3.Y) * (p1.Y - p3.Y));
    double d2 = Math.Sqrt((p2.X - p4.X) * (p2.X - p4.X) +
                          (p2.Y - p4.Y) * (p2.Y - p4.Y));

    return (int)(d1 * d2) / 2;

}

public int triangleArea(Point p1, Point p2, Point p3)
{
    int area = Math.Abs(p1.X * (p3.Y - p2.Y) + p2.X * (p1.Y - p3.Y) +
                        p3.X * (p2.Y - p1.Y)) / 2;
    return area;
}

public override void resizeUpdate()
{
    int min_X = int.MaxValue;
    int min_Y = int.MaxValue;
    int max_X = 0;
    int max_Y = 0;
    updateAllPoints(min_X, min_Y, max_X, max_Y);
    base.setContPoints(rmLocs);
    base.setCenter(this.rmCenter);
}

//the point is in triangle area [p0, p1, p3] or in triangle area [p2, p1, p3]
public override bool isPointInside(Point pnt)
{
    if ((triangleArea(rmLocs[0], rmLocs[1], rmLocs[3]) ==
            triangleArea(pnt, rmLocs[1], rmLocs[3]) +
            triangleArea(rmLocs[0], pnt, rmLocs[3]) +
            triangleArea(rmLocs[0], rmLocs[1], pnt)) ||
        (triangleArea(rmLocs[2], rmLocs[1], rmLocs[3]) ==
            triangleArea(pnt, rmLocs[1], rmLocs[3]) +
            triangleArea(rmLocs[2], pnt, rmLocs[3]) +
            triangleArea(rmLocs[2], rmLocs[1], pnt)))
    {
        return true;
    }
    else
    {
        return false;
    }
}

public override void updateCenter()
{
    this.rmCenter = new Point(rmLocs[1].X, rmLocs[2].Y);
}

public override void Draw(Graphics gr)
{
    gr.DrawPolygon(getlinePen(), rmLocs);
    gr.FillPolygon(new SolidBrush(getFillColor()), rmLocs);
```

```
            base.DrawObjectName(gr);
        }

        public override void rotate()
        {
            Point p1;
            Point p2;
            Point p3;
            Point p4;

            if (Math.Abs(rmLocs[0].X - rmLocs[2].X) > Math.Abs(
                                                    rmLocs[1].Y - rmLocs[3].Y))
            {
                p1 = new Point((rmCenter.X - Math.Abs(
                                rmLocs[1].Y - rmLocs[3].Y) / 2), rmCenter.Y);
                p2 = new Point(rmCenter.X, rmCenter.Y - Math.Abs(
                                rmLocs[0].X - rmLocs[2].X) / 2);
                p3 = new Point((rmCenter.X + Math.Abs(
                                rmLocs[1].Y - rmLocs[3].Y) / 2), rmCenter.Y);
                p4 = new Point(rmCenter.X, rmCenter.Y + Math.Abs(
                                rmLocs[0].X - rmLocs[2].X) / 2);
            }
            else
            {
                p1 = new Point((rmCenter.X + Math.Abs(
                                rmLocs[1].Y - rmLocs[3].Y) / 2), rmCenter.Y);
                p2 = new Point(rmCenter.X, rmCenter.Y + Math.Abs(
                                rmLocs[0].X - rmLocs[2].X) / 2);
                p3 = new Point((rmCenter.X - Math.Abs(
                                rmLocs[1].Y - rmLocs[3].Y) / 2), rmCenter.Y);
                p4 = new Point(rmCenter.X, rmCenter.Y - Math.Abs(
                                rmLocs[0].X - rmLocs[2].X) / 2);
            }
            rmLocs[0] = p1;
            rmLocs[1] = p2;
            rmLocs[2] = p3;
            rmLocs[3] = p4;

            //update base
            base.setContPoints(this.rmLocs);
            base.setCenter(this.rmCenter);
        }

        public override void  move()
        {
            this.rmLocs = base.getContPoints();
            this.rmCenter = base.getCenter();
        }

        public override int ofType()
        {
            return 3;
        }

        public override void WriteXML(XmlDocument doc, XmlElement elem)
        {
            elem = doc.CreateElement("rhombus");
            base.WriteXML(doc, elem);
            doc.DocumentElement.AppendChild(elem);
        }
    }
}
```

Class Triangle – Triangle.cs

```
    public class Triangle : Drawing
    {
        public Point[] trLocs;
        public Point centroid;
        double trArea;

        public Triangle(XmlElement el) : base(3)
        {

            centroid = new Point(Int32.Parse(
               el.GetElementsByTagName("center")[0].Attributes["x"].Value),
               Int32.Parse(
               el.GetElementsByTagName("center")[0].Attributes["y"].Value));

            //load points
            trLocs = new Point[3];
            int i = 0;
            foreach (XmlElement pt_el in el.GetElementsByTagName(
                                            "point_list")[0].ChildNodes)
            {
                trLocs[i] = new Point(Int32.Parse(pt_el.Attributes["x"].Value),
                      Int32.Parse(pt_el.Attributes["y"].Value));
                i++;
            }
            trArea = triangleArea(trLocs[0], trLocs[1], trLocs[2]);
            updateCenter(); //set Centroid
            base.setContPoints(trLocs);
            base.setCenter(this.centroid);
        }


        public Triangle(Point[] cPoints, int id) : base(id, 3)
        {
            this.trLocs = (Point[])cPoints.Clone();
            updateCenter(); //set Centroid
            base.setContPoints(trLocs);
            base.setCenter(this.centroid);
        }

        public Triangle(Point start, Point end, int id) : base(id, 3)
        {
            trLocs = new Point[3];
            this.trLocs[0] = new Point(start.X, start.Y);
            this.trLocs[1] = new Point(start.X, end.Y);
            this.trLocs[2] = new Point(end.X, start.Y);
            trArea = triangleArea(trLocs[0], trLocs[1], trLocs[2]);
            updateCenter(); //set Centroid
            base.setContPoints(trLocs);
            base.setCenter(this.centroid);
        }


        public Triangle(Point start, Point end, Color fill, int id)
            : base(fill, id, 3)
        {
            trLocs = new Point[3];
            this.trLocs[0] = new Point(start.X, start.Y);
            this.trLocs[1] = new Point(start.X, end.Y);
            this.trLocs[2] = new Point(end.X, start.Y);
            trArea = triangleArea(trLocs[0], trLocs[1], trLocs[2]);
            updateCenter(); //set Centroid
            base.setContPoints(trLocs);
            base.setCenter(this.centroid);
        }

        public override void resizeUpdate()
        {
            base.setContPoints(this.trLocs);
            updateCenter();
            base.setCenter(this.centroid);
        }

        public override void updateCenter()
        {
            double ox = trLocs[0].X + trLocs[1].X + trLocs[2].X;
```

```
            double oy = trLocs[0].Y + trLocs[1].Y + trLocs[2].Y;

            this.centroid = new Point((int)ox / 3, (int)oy / 3);
        }

        public override void Draw(Graphics gr)
        {
            gr.DrawPolygon(getlinePen(), trLocs);
            gr.FillPolygon(new SolidBrush(getFillColor()), trLocs);
            base.DrawObjectName(gr);
        }

        public int triangleArea(Point p1, Point p2, Point p3)
        {
            int area = Math.Abs(p1.X * (p3.Y - p2.Y) + p2.X * (p1.Y - p3.Y) +
                       p3.X * (p2.Y - p1.Y)) / 2;
            return area;
        }

        public override bool isPointInside(Point pnt)
        {
            if (triangleArea(trLocs[0], trLocs[1], trLocs[2]) ==
                      triangleArea(pnt, trLocs[1], trLocs[2]) +
                      triangleArea(trLocs[0], pnt, trLocs[2]) +
                      triangleArea(trLocs[0], trLocs[1], pnt))
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        void rotate(ref Point point, Point origin, double angle)
        {
            int X = origin.X + (int) ((point.X - origin.X) * Math.Cos(angle) -
                    (point.Y - origin.Y) * Math.Sin(angle));
            int Y = origin.Y + (int) ((point.X - origin.X) * Math.Sin(angle) +
                    (point.Y - origin.Y) * Math.Cos(angle));
            point = new Point(X, Y);
        }

        void rotatePolygon(Point[] points, int number, Point origin, double angle)
        {
            for (int i = 0; i < number; i++)
                rotate(ref points[i], origin, angle);
        }


        public override void rotate()
        {
            rotatePolygon(trLocs, 3, centroid, Math.PI/2);

            //update base
            base.setContPoints(this.trLocs);
            base.setCenter(this.centroid);

        }

        public override void move()
        {
            this.trLocs = base.getContPoints();
            this.centroid = base.getCenter();
        }

        public override int ofType()
        { return 2; }

        public override void WriteXML(XmlDocument doc, XmlElement elem)
        {
            elem = doc.CreateElement("triangle");
            base.WriteXML(doc, elem);
            doc.DocumentElement.AppendChild(elem);
        }
    }
```

Class DrawConnection – DrawConnection.cs

```
    public class DrawConnection
    {
        string name;
        DrawVertex Source;
        DrawVertex Dest;
        bool isDirect;
        double Weight;
        Pen shPen;
        Pen arrowPen;
        public const string regSepartor = "$$$";
        public const string dirSepartor = "-->";


        public DrawConnection()
        {
            this.Source = null;
            this.Dest = null;
            this.isDirect = false;
            this.Weight = 0;
            this.name = "";
            this.shPen = new Pen(Color.SeaGreen);
            this.arrowPen = new Pen(Color.Chocolate);
        }


        public void FormDrawConnectionName()
        {
            if (!this.isDirect)
            {
                this.name = this.Source.getName() + regSepartor + this.Dest.getName();
            }
            else
            {
                this.name = this.Source.getName() + regSepartor +
                            this.Dest.getName() + dirSepartor;
            }
        }

        public string getName()
        {
            return this.name;
        }

        public void setIsDirect(bool isD)
        {
            this.isDirect = isD;
        }

        public bool getIsDirect()
        {
            return this.isDirect;
        }

        public string getSourceName()
        {
            return this.Source.getName();
        }

        public DrawVertex getSource()
        {
            return this.Source;
        }

        public int getSourceId()
        {
            return this.Source.getId();
        }


        public string getDestName()
        {
            return this.Dest.getName();
        }

        public int getDestId()
```

```
        {
            return this.Dest.getId();
        }

        public DrawVertex getDest()
        {
            return this.Dest;
        }

        public void setWeight(double d)
        {
            this.Weight = d;
        }

        public double getWeight()
        {
            return this.Weight;
        }

        public DrawConnection(DrawVertex s, DrawVertex d, bool isD)
        {
            this.Source = s;
            this.Dest = d;
            this.isDirect = isD;
            this.Weight = 0;
            this.name = "";
            this.shPen = new Pen(Color.SeaGreen);

            this.arrowPen = new Pen(Color.Black, 4);
            this.arrowPen.SetLineCap(LineCap.Flat, LineCap.ArrowAnchor, DashCap.Flat);
            FormDrawConnectionName();

        }

        public DrawConnection(DrawVertex s, DrawVertex d, bool isD, string nm)
        {
            this.Source = s;
            this.Dest = d;
            this.isDirect = isD;
            this.Weight = 0;
            this.name = nm;
            this.shPen = new Pen(Color.SeaGreen);
        }

        public void drawDrawConnection(Graphics gr)
        {
            Point d = Dest.getCenter();
            Point s = Source.getCenter();
            int hypotenuse = (int)Math.Sqrt((d.X - s.X) * (d.X - s.X) +
                                            (s.Y - d.Y) * (s.Y - d.Y));

            //case self-pointing DrawConnection
            if (d.X == s.X && d.Y == s.Y)
            {
                return;
            }

            int alpha_x = Dest.getRadius() * (d.X - s.X) / hypotenuse;
            int alpha_y = Dest.getRadius() * (s.Y - d.Y) / hypotenuse;

            if (!this.isDirect)
            {
                gr.DrawLine(shPen, new Point(s.X + alpha_x, s.Y - alpha_y),
                                   new Point(d.X - alpha_x, d.Y + alpha_y));
            }
            else
            {
                gr.DrawLine(arrowPen, new Point(s.X + alpha_x, s.Y - alpha_y),
                                      new Point(d.X - alpha_x, d.Y + alpha_y));


            }
        }

    }
```

Class DrawVertex – DrawVertex.cs

```
    public class DrawVertex
    {
        private string Name;
        private int Id;
        int Radius;
        Point Center;
        Font nFont;
        Brush nBrush;
        Pen shPen;

        bool isActive;

        public DrawVertex(Point cent)
        {
            Radius = 20;
            Name = "";
            Id = 0;
            Center = cent;
            shPen = new Pen(Color.Black);

            nFont = new Font("Arial", 14, GraphicsUnit.Pixel);
            nBrush = new SolidBrush(Color.Blue);
            isActive = false;
        }

        public DrawVertex(Point cent, string nm, int id)
        {
            Radius = 20;
            Name = nm;
            Id = id;
            Center = cent;
            shPen = new Pen(Color.Black);

            nFont = new Font("Arial", 14, GraphicsUnit.Pixel);
            nBrush = new SolidBrush(Color.Blue);
            isActive = false;
        }

        public DrawVertex(DrawVertex cir)
        {
            this.Radius = cir.Radius;
            this.Name = cir.Name;
            this.Id = cir.Id;
            this.Center = cir.Center;
            this.shPen = cir.shPen;
            this.nFont = cir.nFont;
            this.nBrush = cir.nBrush;
            isActive = false;
        }

        public void setId(int id)
        {
            this.Id = id;
        }

        public int getId()
        {
            return this.Id;
        }

        public void setCenter(Point pnt)
        {
            Center = pnt;
        }

        public void setCenter(int x, int y)
        {
            Center = new Point(x, y);

        }

        public Point getCenter()
        {
            return Center;
        }
```

```
    public void setName(string nm)
    {
        Name = nm;
    }

    public string getName()
    {
        return Name;
    }

    public int getRadius()
    {
        return Radius;
    }

    public void setActive(bool a)
    {
        isActive = a;
    }

    public bool getActive()
    {
        return isActive;
    }

    public void drawActive(Graphics gr)
    {
        Pen activePen = new Pen(Color.Red);

        gr.DrawEllipse(activePen, Center.X - Radius, Center.Y - Radius,
                       2 * Radius, 2 * Radius);
        gr.FillEllipse(new SolidBrush(Color.White), Center.X - Radius,
                       Center.Y - Radius, 2 * Radius, 2 * Radius);

        double cnt = Name.Length;
        int strWidth = (int)gr.MeasureString(Name, nFont).Width;
        int strHight = (int)gr.MeasureString(Name, nFont).Height;
        gr.DrawString(Name, nFont, nBrush,
             new Point(Center.X - strWidth / 2, Center.Y - strHight / 2));
    }


    public void drawDrawVertex(Graphics gr)
    {
        gr.DrawEllipse(shPen, Center.X - Radius, Center.Y - Radius,
                       2 * Radius, 2 * Radius);
        gr.FillEllipse(new SolidBrush(Color.LightCoral), Center.X - Radius,
                        Center.Y - Radius, 2 * Radius, 2 * Radius);
    }

    public void drawDrawVertexWithName(Graphics gr)
    {
        gr.DrawEllipse(shPen, Center.X - Radius, Center.Y - Radius,
                       2 * Radius, 2 * Radius);
        if (isActive == true)
        {
            gr.FillEllipse(new SolidBrush(Color.Red), Center.X - Radius,
                       Center.Y - Radius, 2 * Radius, 2 * Radius);
        }
        else
        {
            gr.FillEllipse(new SolidBrush(Color.LightCoral), Center.X - Radius,
                        Center.Y - Radius, 2 * Radius, 2 * Radius);
        }


        double cnt = Name.Length;
        int strWidth = (int)gr.MeasureString(Name, nFont).Width;
        int strHight = (int)gr.MeasureString(Name, nFont).Height;
        gr.DrawString(Name, nFont, nBrush,
                    new Point(Center.X - strWidth / 2, Center.Y - strHight / 2));
    }

}
```

Class GraphVertex – GraphVertex.cs

```
    public class GraphVertex
    {
        int id;
        string name;
        Dictionary<int, double> neighbors;

        public GraphVertex(int id, string name, Dictionary<int, double> neighbors)
        {
            this.id = id;
            this.name = name;
            this.neighbors = neighbors;
        }

        public GraphVertex(int id, Dictionary<int, double> neighbors)
        {
            this.id = id;
            this.name = "";
            this.neighbors = neighbors;
        }

        public GraphVertex(int id, List<int> neighbor_ids)
        {
            this.id = id;
            this.name = "";
            this.neighbors = new Dictionary<int, double>();
            foreach (int cur in neighbor_ids)
            {
                this.neighbors.Add(cur, 0.0);
            }
        }

        public GraphVertex(int id, string name, List<int> neighbor_ids)
        {
            this.id = id;
            this.name = name;
            this.neighbors = new Dictionary<int, double>();
            foreach (int cur in neighbor_ids)
            {
                this.neighbors.Add(cur, 0.0);
            }
        }

        public GraphVertex(int id)
        {
            this.id = id;
            this.name = "";
            this.neighbors = new Dictionary<int, double>();
        }

        public int getId()
        {
            return this.id;
        }

        public void setID(int id)
        {
            this.id = id;
        }

        public string getName()
        {
            return this.name;
        }

        public void setName(string name)
        {
            this.name = name;
        }

        public Dictionary<int, double> getNeighbors()
        {
            return this.neighbors;
        }
```

```
    public void setNeighbors(Dictionary<int, double> neighbors)
    {
        this.neighbors = neighbors;
    }
}
```

Class MyEnumerator – MyEnumerator.cs

```
    // Declare the enumerator class:
    public class MyEnumerator
    {
        int nIndex;
        Graph collection;
        public MyEnumerator(Graph coll)
        {
            collection = coll;
            nIndex = -1;
        }

        public bool MoveNext()
        {
            nIndex++;
            return (nIndex < collection.getVertexes().Count);
        }

        public GraphVertex Current
        {
            get
            {
                return (collection.getVertexes()[nIndex]);
            }
        }
    }
```

Class Graph – Graph.cs

```
    public class Graph
    {
        List<GraphVertex> Vertexes;
        public Graph( ArrayList drVertexes, ArrayList drConnections)
        {
            //the global vertexes parameter
            Vertexes = new List<GraphVertex>();

            //Temp variable vertex to be added to vertexes
            GraphVertex vertex;

            //Temp list of drawConnections to be assigned to current vertex
            List<int> drConn;

            foreach (DrawVertex dv in drVertexes)
            {
                drConn = new List<int>();
                foreach(DrawConnection dc in drConnections)
                {
                    if (dc.getSourceId() == dv.getId())
                    {
                    drConn.Add(dc.getDestId());
                    }
                }

                vertex = new GraphVertex(dv.getId(), dv.getName(), drConn);
                Vertexes.Add(vertex);
            }

        }

        public MyEnumerator GetEnumerator()
        {
            return new MyEnumerator(this);
        }


        public List<GraphVertex> getVertexes()
        {
            return Vertexes;
        }

    }
```

Class GraphVertex – GraphVertex.cs

```
    public class GraphVertex
    {
        int id;
        string name;
        Dictionary<int, double> neighbors;

        public GraphVertex(int id, string name, Dictionary<int, double> neighbors)
        {
            this.id = id;
            this.name = name;
            this.neighbors = neighbors;
        }

        public GraphVertex(int id, Dictionary<int, double> neighbors)
        {
            this.id = id;
            this.name = "";
            this.neighbors = neighbors;
        }

        public GraphVertex(int id, List<int> neighbor_ids)
        {
            this.id = id;
            this.name = "";
            this.neighbors = new Dictionary<int, double>();
            foreach (int cur in neighbor_ids)
            {
                this.neighbors.Add(cur, 0.0);
            }
        }

        public GraphVertex(int id, string name, List<int> neighbor_ids)
        {
            this.id = id;
            this.name = name;
            this.neighbors = new Dictionary<int, double>();
            foreach (int cur in neighbor_ids)
            {
                this.neighbors.Add(cur, 0.0);
            }
        }

        public GraphVertex(int id)
        {
            this.id = id;
            this.name = "";
            this.neighbors = new Dictionary<int, double>();
        }

        public int getId()
        {
            return this.id;
        }

        public void setID(int id)
        {
            this.id = id;
        }

        public string getName()
        {
            return this.name;
        }

        public void setName(string name)
        {
            this.name = name;
        }

        public Dictionary<int, double> getNeighbors()
        {
            return this.neighbors;
        }

        public void setNeighbors(Dictionary<int, double> neighbors)
```

```
    {
        this.neighbors = neighbors;
    }
}

// Declare the enumerator class:
public class MyEnumerator
{
    int nIndex;
    Graph collection;
    public MyEnumerator(Graph coll)
    {
        collection = coll;
        nIndex = -1;
    }

    public bool MoveNext()
    {
        nIndex++;
        return (nIndex < collection.getVertexes().Count);
    }

    public GraphVertex Current
    {
        get
        {
            return (collection.getVertexes()[nIndex]);
        }
    }
}



public class Graph
{
    List<GraphVertex> Vertexes;
    public Graph( ArrayList drVertexes, ArrayList drConnections)
    {
        //the global vertexes parameter
        Vertexes = new List<GraphVertex>();

        //Temp variable vertex to be added to vertexes
        GraphVertex vertex;

        //Temp list of drawConnections to be assigned to current vertex
        List<int> drConn;

        foreach (DrawVertex dv in drVertexes)
        {
            drConn = new List<int>();
            foreach(DrawConnection dc in drConnections)
            {
                if (dc.getSourceId() == dv.getId())
                {
                drConn.Add(dc.getDestId());
                }
            }

            vertex = new GraphVertex(dv.getId(), dv.getName(), drConn);
            Vertexes.Add(vertex);
        }

    }

    public MyEnumerator GetEnumerator()
    {
        return new MyEnumerator(this);
    }


    public List<GraphVertex> getVertexes()
    {
        return Vertexes;
    }

}
```

References

1. L. Panait, S. Luke (2005). Cooperative Multi-Agent Learning: The State of the Art. Autonomous Agents and Multi-Agent Systems 11(3), pp. 387-434

2. S. Russell, P. Norvig (2003). Artificial Intelligence: A Modern Approach (2nd ed.). Upper Saddle River, NJ: Prentice Hall

3. A. Garland and R. Alterman (2004). Autonomous agents that learn to better coordinate. Autonomous Agents and Multi-Agent Systems, Vol. 8, pp. 267–301

4. C. Goldman and J. Rosenschein (1996). Mutually supervised learning in multiagent systems. Adaptation and Learning in Multi-Agent Systems, pp. 85–96

5. A. Barto, R. Sutton, and C. Watkins (1990). Learning and sequential decision making. In M. Gabriel and J. Moore, Learning and computational neuroscience : foundations of adaptive networks, M.I.T. Press

6. L. Kaelbling, M. Littman, and A. Moore (1996). Reinforcement learning: A survey. Journal of Artificial Intelligence Research, Vol. 4, pp. 237–285

7. T. Haynes, K. Lau, and S. Sen (1996). Learning cases to compliment rules for conflict resolution in multiagent systems. AAAI Spring Symposium on Adaptation, Coevolution, and Learning in Multiagent Systems, pages 51–56

8. T. Haynes, S. Sen (1997). Crossover operators for evolving a team. Genetic Programming 1997: Proceedings of the Second Annual Conference, pages pp.162–167

9. T. Haynes, S. Sen, D. Schoenefeld, R.Wainwright [a](1995). Evolving a team. Working Notes for the AAAI Symposium on Genetic Programming, pp 23–30

10. T. Haynes, S. Sen, D. Schoenefeld, R.Wainwright [b](1995). Evolving multiagent coordination strategies with genetic programming. Technical Report UTULSA-MCS-95-04, The University of Tulsa.

11. H. Iba. (1996) Emergent cooperation for multiple agents using genetic programming. Parallel Problem Solving from Nature IV: Proceedings of the International Conference on Evolutionary Computation, Vol. 1141, pp 32–41

12. H. Iba (1998). Evolutionary learning of communicating agents. Information Sciences, Vol. 108

13. T. Miconi (2003). When evolving populations is better than coevolving individuals: The blind mice problem. In Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03).

14. S. Abney (2008). Semisupervised Learning for Computational Linguistics. Chapman & Hall/CRC.

15. Y. Altun, D. McAllester, M. Belkin (2005). Maximum margin semi-supervised learning for structured variables. Advances in Neural Information Processing Systems (NIPS) Vol. 18.

16. A. Blum, T. Mitchell (1998). Combining labeled and unlabeled data with co-training. COLT: Proceedings of the Workshop on Computational Learning Theory, Morgan Kaufmann, pp. 92-100.

17. X. Zhu (2005) [a]. Semi-Supervised Learning with Graphs, Doctoral thesis, CMU-LTI-05-192

18. J. Zhu. 2005 [b] (2005). Semi-supervised learning literature survey. Computer Sciences Technical Report TR 1530.

19. E. Feigenbaum (1961). The simulation of verbal learning behavior. Proceedings of the Western Joint Computer Conference, Vol. 19, pp. 121–131.

20. T. Mitchell (1977). Version spaces: A candidate elimination approach to rule learning. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77), pp. 305–310

21. T. Mitchell (1982). Generalization as search. Artificial Intelligence, Vol. 18(2), 203–226.

22. B. Buchanan, T. Mitchell, R. Smith, C. Johnson (1978). Models of learning systems. Encyclopedia of Computer Science and Technology, Vol. 11.

23. T. Mitchell, P. Utgoff, R. Banerji, (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. Morgan Kaufmann, San Mateo, California.

24. R. Fikes, P. Hart, N. Nilsson (1972). Learning and executing generalized robot plans. Artificial Intelligence, Vol. 3(4), pp. 251–288.

25. S. Minton (1988). Quantitative results concerning the utility of explanation-based learning. In Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88), pp. 564–569

26. P. Tadepalli (1993). Learning from queries and examples with tree-structured bias. In Proceedings of the Tenth International Conference on Machine Learning, pp. 322–329

27. J. Quinlan (1990). Learning logical definitions from relations. Machine Learning, Vol. 5(3), pp. 239–266

28. S. Muggleton and W. Buntine (1988). Machine invention of first-order predicates by inverting resolution. In Proceedings of the Fifth International Conference on Machine Learning, pp. 339–352

29. S. Muggleton (1995). Inverse entailment and Progol. New Generation Computing, Special issue on Inductive Logic Programming, Vol. 13(3-4), pp. 245–286.

30. S. Muggleton (2000). Learning stochastic logic programs. Proceedings of the AAAI 2000 Workshop on Learning Statistical Models from Relational Data.

31. A. Bryson, Y. Ho (1969). Applied Optimal Control. Blaisdell, New York.

32. P. Werbos (1974). Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. Ph.D. thesis, Harvard University, Cambridge, Massachusetts.

33. D. Parker (1985). Learning logic. Technical report TR-47, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, Massachusetts.

34. J. Hopfield (1982). Neurons with graded response have collective computational properties like those of two-state neurons. Proceedings of the National Academy of Sciences of the United States of America, Vol. 79, pp. 2554–2558.

35. V. Vapnik (1998). Statistical Learning Theory. Wiley, New York.

36. N. Cristianini and B. Scholkopf (2002). Support vector machines and kernel methods: The new generation of learning machines. AI Magazine, Vol. 23(3), pp. 31–41.

37. B. Scholkopf, A. Smola (2002). Learning with Kernels. MIT Press, Cambridge, Massachusetts.

38. T. Joachims (2001). A statistical learning model of text classification with support vector machines. In Proceedings of the 24th Conference on Research and Development in Information Retrieval (SIGIR), pp. 128–136

39. M. Brown, W. Grundy, D. Lin, N. Cristianini, C. Sugnet, T. Furey, M. Ares, D. Haussler (2000). Knowledge-based analysis of microarray gene expression data using support vector machines. In Proceedings of the national Academy of Sciences, Vol. 97, pp. 262–267

40. D. DeCoste and B. Scholkopf (2002). Training invariant support vector machines. Machine Learning, Vol. 46(1), pp. 161–190.

41. R. Sutton, D. McAllester, S. Singh, Y. Mansour (2000) Policy Gradient Methods for Reinforcement Learning with Function Approximation. Advances in Neural Information Processing Systems Vol. 12, pp. 1057-1063

42. A. da Motta Salles Barreto, C. Anderson (2008). Restricted gradient-descent algorithm for value-function approximation in reinforcement learning. Artificial Intelligence Volume 172 Issue 4-5

43. S. Whiteson, P. Stone (2006). Evolutionary Function Approximation for Reinforcement Learning. The Journal of Machine Learning Research, Volume 7

44. C. Ribeiro (2002). Reinforcement Learning Agents. Artificial Intelligence Review , Volume 17(3)

45. M. Ghavamzadeh, S. Mahadevan (2007). Hierarchical Average Reward Reinforcement Learning. The Journal of Machine Learning Research , Volume 8

46. H. Fujita, S. Ishii (2009). Model-Based Reinforcement Learning for Partially Observable Games with Sampling-Based State Estimation. Neural Computation , Volume 19 (11)

47. Y. Duan, Q. Liu, X. Xu (2007). Application of reinforcement learning in robot soccer. Engineering Applications of Artificial Intelligence, Volume 20 (7)

48. J. Quinlan (1993). C4.5: Programs for machine learning. Morgan Kaufmann. San Mateo, CA.

49. S. Murthy (1998). Automatic construction of decision trees from data: a multidisciplinary survey. Data Mining and Knowledge Discovery, 2, pp.345–389.

50. P. Clark, R. Boswell (1989). The CN2 induction algorithm. Machine Learning, 3 (4), pp. 261–283.

51. J. Han, M. Kamber (2006). Data mining: concept and techniques". San Francisco, CA: Morgan Kaufmann.

52. J. Shafer, R. Agrawal, M. Mehta (1996). A scalable parallel classifier for data mining. International Conference on Very Large Databases, pp. 544–555.

53. N. Nilsson (1996). Book "Introduction to Machine Learning"

54. R. Sutton, A. Barto (1998). Book "Reinforcement Learning: An introduction". MIT Press, Cambridge, MA.

55. A. Doniec, R. Mandiau, S. Piechowiak, S. Espié (2008). A behavioral multi-agent model for road traffic simulation Engineering. Applications of Artificial Intelligence , Volume 21(8)

56. B. Montano, V. Yoon, K. Drummey, J. Liebowitz (2008). Agent learning in the multi-agent contracting system [MACS]. Decision Support Systems , Volume 45(1)

57. R. Stanley (1973). Acyclic orientations of graphs, Discrete Mathematics 5, pp 171-178

58. J. Bang-Jensen (2008). 2.1 Acyclic Digraphs, Digraphs: Theory, Algorithms and Applications, Springer Monographs in Mathematics (2nd ed.), Springer-Verlag, pp. 32–34

59. A. Czumaj, M. Kowaluk, A. Lingas (2007). Faster algorithms for finding lowest common ancestors in directed acyclic graphs. Theoretical Computer Science, Vol .380 (1-2), pp.37-46

60. D. Pearce, P. Kelly (2006). A dynamic topological sort algorithm for directed acyclic graphs. Journal of Experimental Algorithmics (JEA) Vol. 11

61. R. Bellman, S. Dreyfus (1962) Applied Dynamic Programming. Princeton University Press.

62. R. Bird, O. Moor (1993). From dynamic programming to greedy algorithms. State-of-the-Art Seminar on Formal Program Development, Springer LNCS 755.

63. W. Powell (2007). Approximate Dynamic Programming: Solving the Curses of Dimensionality. Wiley Series in Probability and Statistics, Wiley-Interscience