*Proceedings of the 8th International Conference on Applied Informatics*
*Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 217–225.*

# Modeling Dynamic Programming Problems: Petri Nets Versus d-graphs

**Zoltán Kátai, Péter István Fülöp**

Sapientia University, Tirgu Mures / Corunca, Romania

**Abstract**

Dynamic programming (DP) is a general optimization technique, which can be applied to numerous decision problems that typically require a sequence of decisions to be made. The most common approach taken today for solving real-world DP problems is to start a specialized software development project for every problem in particular. Even taking into account that a module-oriented development philosophy allows considerable reuse of components, this is still a rather expensive approach. Consequently, a general software tool that automatically solves DP problems (getting as input the functional equation) should be able to save considerable software development costs.

Lew and Mauch [1, 2] introduced Bellman nets (special high level colored Petri nets) as modeling tools for discrete DP problems, and Mauch [3] developed the solver software DP2PN2solver that builds Bellman nets as intermediate representation of the functional equation. On the other hand Kátai [4, 5] introduced d-graphs for modeling DP problems, and Kátai and Csíki [6] developed a d-graph oriented DP solver software tool. In this paper we compare the two methods and software tools.

## 1. Introduction

Dynamic programming (DP) is a general optimization technique, which can be applied to numerous decision problems that typically require a sequence of decisions to be made. The most common approach taken today for solving real-world DP problems is to start a specialized software development project for every problem in particular. Even taking into account that a module-oriented development philosophy allows considerable reuse of components, this is still a rather expensive approach. Consequently, a general software tool that automatically solves DP problems (getting as input the functional equation) should be able to save considerable software development costs.

So far, software tools that would allow the user to conveniently state arbitrary DP problems using the terminology and techniques that have been established in the past in the field of DP have not been developed. One of the difficulties in designing a DP solver system is to come up with a specification language that is on the one hand general enough to capture the majority of DP problems arising in reality, and on the other hand structured enough to be parsed efficiently. For linear programming (LP) problems, it is very easy to achieve both of these goals, for DP problems however it is much harder to satisfy these conflicting goals. An intermediate problem representation in the form of a Petri net (PN) or d-graph model turns out to be a useful device for this purpose. The mentioned models are only applicable to "discrete" optimization problems for which a DP functional equation with a ?nite number of states and decisions can be obtained.

Mauch proposed a specialized PN model that uses the standard semantics of place/transition nets, a low-level PN class [3], whereas Lew's model relies on high-level PNs with numerically-colored tokens, called Bellman Nets [1]. D-graphs as modeling tools for DP problems were introduced by Katai in [4]. Mauch [3] developed the solver software DP2PN2solver that builds Bellman nets as intermediate representation of the functional equation, and Kátai and Csíki [6] developed a d-graph oriented DP solver software tool. In this paper we compare the two methods and software tools.

## 2. Modeling DP problems

As we mentioned above, DP is often used to solve optimizing problems. The problem usually consists of a target function, which has to be optimized through an optimal sequence of decisions. DP is built on the principle of optimality: the optimal solution is built by optimal sub-solutions. This principle is expressed by a recursive formula (functional equation), which describes mathematically the way the more and more complex optimal sub-solutions are built from the simpler ones. Obviously, this is a formula where the way of the optimal (minimum or maximum) decision making has been built in. Once the functional equation is established, the problem can be considered mathematically solved.

The programming part of the problem solving process is built on another principle of DP, which is: the optimal values of the target function concerning the already solved sub-problems are stored (often in an array). According to the principle of the optimality we are interested only in the optimal solutions of the sub-problems. This technique, often called memoization or result catching, makes it possible to avoid the repeating computation for overlapped sub-problems, which are also characteristic for DP problems. The core of the computer program that implements the DP algorithm consists in computing the corresponding elements of the array in bottom-up way according to the strategy given by the recursive formula. An efficient strategy solves each sub-problem before its optimum value is needed by any other sub-problem. The complexity of this programming task varies from problem to problem. It is often nontrivial to write a code that evaluates the sub-problems

in the most efficient order.

Suppose a discrete optimization problem can be solved by an integer dynamic programming equation of the form

$$f(s) = \min_a \left\{ d(a) + \sum_{k=1}^{K_a} \left( f(s^{(k)}) \right) \right\},$$

with nonnegative integral base cases values $f(s_0)$ given. Here s denotes a state, f is the minimization functional, a is a decision that can be made in state $s$, $d$ is the decision cost function, and $s(k)$ are the $K_a$ next states, and $s_0$ are base case (initial condition) states.

## 2.1. Construction of the PN Model

Familiarity with basic Petri net terminology is assumed in this paper. The PN model corresponding to the above DP problem has

1. a state place $p_s$ for each state s, that has, except for the base case state places, a unique minimization transition in its preset,
2. a min-in place $p_m$ for each decision a, that has a minimization transition in its postset, and whose initial marking equals d(a),
3. a minimization transition associated with each state place $p_s$, except for the base case state places, that has in its preset the min-in places for each decision a,
4. a copy transition that transfers tokens from the state place associated with s(k) to those min-in places whose decision a involves the summand $f(s(k))$.

Transitions serve two different purposes in the model:

- *Processing Transitions:* Transitions can be considered as processing elements. E.g. there are minimization transitions, addition transitions, multiplication transitions, etc. In other words there are transitions representing arbitrary functions f.
- *Copy Transitions:* Another purpose of transitions is to make the results of solved subproblems available to the super-problems requesting these results. Each of these special "copy transitions" distributes a result from one "state place" to multiple "min-in places".

There are two types of places in the PN model:

- *State Places:* Such a place represents a state encountered during the solution process of a DP problem. Except for the base case state places, state places are the output places for minimization transitions.

- *Min-in Places:* These are the input places for minimization transitions. They are also the output places for copy transitions.

The role of markings in the PN model:

- *Marking of State Places:* Let $p_s$ be a state place. Immediately after all minimization transition in its preset $\bullet p_s$ have fired, but just before firing any of the copy transitions in its postset $p_s\bullet$, the marking of a state place $p_s$ representing state s contains f(s) tokens, representing the optimal value of the subproblem associated with state s. Base case state places are initially marked with an appropriate number of tokens as specified in the base cases of the DP functional equation. That is a base case state place associated with a base case state $s_0$ is initially marked with $f(s_0)$ tokens. All other state places are initially marked with no tokens.

- *Marking of Min-in Places:* Let $p_m$ be a min-in place. After all transitions in its preset $\bullet p_m$ have fired, but just before firing any transitions in its postset $p_m\bullet$, the marking of these places corresponds to the values to be minimized by a minimization transition. Min-in places are initially marked with d(a) tokens.

## 2.2. The Intermediate Bellman Net Representation

A Bellman Net is a special high-level colored Petri net with the following properties.

1. The color type is numerical in nature, tokens are real numbers. In addition, single black tokens are used to initialize enabling places, which are technicalities that prevent transitions from firing more than once.

2. A place contains at most one token at any given time.

3. The postset of a transition contains exactly one designated output place, which contains the result of the computational operation performed by the transition.

4. There are several different types of transitions, among them M-transitions and E-transitions. An M-transition performs a minimization or maximization operation using the tokens of the places in its preset as operands and puts the result into its designated output place. An E-transition evaluates a basic arithmetic expression (involving operators like addition or multiplication) using fixed constants and tokens of the places in its preset as operands and puts the result into its designated output place.

5. There are self-loops between an E-transition and all places in its preset. Their purpose is to conserve operands serving as input for more than one E-transition.

A numerical token as the marking of a place in a Bellman net can be interpreted as an intermediate value that is computed in the course of calculating the solution of a corresponding DP problem instance.

While the expressional power of both (low/high level PN) approaches seems to be equivalent (no proof yet) it is easier to transform Bellman Nets to executable computer code. However, the low level PN model is easier to examine with respect to consistency, and other net theoretic issues.

## 2.3. DP and d-graphs

Considering the recursive formula (functional equation) as an implicit description of a d-graph, DP problems can be interpreted as optimal path/tree problems in the associated d-graph.

**Definition 2.1.** The connected weighted bi-parted digraph $G_d$ (V, E, C) is a d-graph if:

- $V = V_p \square V_d$ and $E = E_p \square E_d$
- $V_p$ - the set of the p-nodes.
- $V_d$ - the set of the d-nodes
- All in/out neighbours of the p-nodes (excepting the source/sink nodes) are d-nodes. Each d-node has exactly one p-out-neighbour. Each d-node has at least one p-in-neighbour.
- $E_p$ - the set of p-arcs. (from d-nodes to p-nodes)
- $E_d$ - the set of d-arcs. (from p-nodes to d-nodes)
- The C: $E_p \rightarrow R$ function associates a cost to every p-arc. We consider the d-arcs of zero cost.

For further definitions (d-subgraph, d-tree, d-subtree, spanning d-subtree, optimal spanning d-subtree, etc.) see [4].

*Modeling DP problems with d-graphs:*

- The p-nodes represent the subproblems. The source-nodes represent the trivial subproblems (associated with the base cases) whereas the sink-node (or nodes) represent the original problem (problem-set).
- $f(s)$ is the weight of the p-node associated with state $s(p_s)$, and the number of its d-in-neighbours is equal with the number of choices $a$ at the decision has to be made in state s. The p-in-arcs of $p_s$ are weighted with cost-values d(a), and its d-in-neighbours with the sums (according to the above example

we have sum-function, but the nature of this function depends on the DP problem to be solved)

$$\sum_{k=1}^{K_a} f(s^{(k)})$$

- The d-in-neighbour corresponding to decision $a$ has $\mathrm{K}_a$ p-in-neighbours representing the subproblems associated with states $s(k)$ $(k = 1, \ldots, K_a)$.
- The optimal solution (the sequence of optimal decision) is represented by the optimal spanning d-subtree of the associated d-graph. The weights of the p-nodes correspond to the optimal objective function values.

The following three cases can be distinguished [5]:

1. The attached d-graph is cycle free. In this case the most efficient optimal path/tree algorithm is based on the topological order of the nodes.
2. The graph contains cycles, but there are no negative weight arcs. For this case the best choice is a d-graph variant of Dijkstra's algorithm.
3. The graph has negative arcs, but it has no negative weight cycles. This optimal path/tree problem is solved by a d-graph variant of Bellman-Ford algorithm.

# 3. DP solver software tools

## 3.1. Solver software DP2PN2solver

Mauch describes in [3] the solver software DP2PN2solver that builds Bellman nets as intermediate representation of the functional equation. The following are the steps that need to be performed to solve a DP problem instance with DP2PN2solver (only step 1 is performed by the human DP modeler whereas all other steps are automatically performed by the software tool).

1. It models the real-world problem and creates the DP specification-file in the gDPS language.
2. The appropriate DP2PN module produces the intermediate Bellman net representation.
3. One of the PN2Solver modules produces runnable Java code, or a spreadsheet or another form of executable solver code, which is capable of solving the problem instance.
4. It runs the resulting executable solver code and outputs the solution of the problem instance.

### 3.2. DP solver software based on intermediate d-graph representation

The core idea of the algorithm behind the software is that we represent explicitly the graph described implicitly by the recursive formula. There are two strategies to transpose the functional equation into an algorithm: the direct method (direct-conversion of the functional equation into an iterative/recursive procedure) and the successive approximation methods (after an initial approximation, the array-cells that are going to store the optimum values are successively updated – improved – either by the functional equation itself or by an equation related to it) [7].

Another classification of the DP strategies is based on the way the optimum values of the sub-problems are computed. The so-called pull-approach computes directly (not by an updating process) the optimum value of the current node on the basis of the already computed optimum values of its immediate predecessors. This approach is an immediate application of the functional equation, and can be used only for the acyclic graphs [7]. The key idea in the case of the push-approach is to propagate any improvement that has been made in the current node to its out-neighbors. The algorithm ends when any other improvements cannot be performed [7]. All the three optimal path/tree algorithms integrated in the software apply successive approximation and push-approach.

*The algorithm is:*
(Only step 1 is performed by the human DP modeler whereas all other steps are automatically performed by the software tool)
1. Input:
   - The recursive formula is introduced.
   - The index-limits (along every dimension) of the array are introduced.
   - The indexes of the cell that represents the original problem are introduced.
2. The recursive formula is analyzed:
   - The software asks for the input data.
   - The d-graph is built.
3. The type of the d-graph is determined. (A DFS algorithm tests if the d-graph is acyclic or not, has negative arcs or not, and whether it contains negative cycles or not.)
4. The proper optimal path/tree algorithm is applied.
5. The solution (the optimum value corresponding to the original problem, and the cell-indexes along the optimal path/tree) is printed.

## 4. Petri Nets versus d-graphs: comparative analysis

1. Both methods apply intermediate representation in order to hide the variety of DP problems.
2. Both methods move the DP problem to be solved to a well research area: Petri nets or graph theory (optimal path/tree algorithms). While DP2PN2solver

integrates standard Petri-algorithms, the optimal path/tree algorithms had to be adapted to d-graphs.

3. Both Petri nets and d-graphs are specialized bi-parted directed graphs.

4. Because of their illustrability both modeling methods are valuable didactic tools in teaching-learning process of dynamic programming.

5. Since d-graphs were explicitly developed as modeling tools for DP problems the attached d-graph is more concise then the corresponding Petri net representation (the roles of M/E transitions are integrated in the weight-functions of the p/d -nodes)

   - Mauch states: in case of problems when after each decision exactly one successor state needs to be evaluated a weighted directed graph representation (states are nodes and edge weights represent the cost of a decision) would be sufficient; when after each decision two or more successor states need to be evaluated a weighted directed graph representation is no longer sufficient. Introducing d-graphs Katai solves this impediment without moving the problem to the field of Petri nets.

6. The use of DP2PN2solver presumes to learn the gDPS language (this is a general source language that can describe a variety of DP problems; it offers the flexibility needed for the various types of DP problems that arise in reality). On the other hand the "d-graph software tool" uses as input format for the functional equation a more traditional form that is closer to its mathematical formulation. (The modeling power of the two input module has not been compared yet)

7. There are DP problems with "cyclic functional equation" (the chain of recursive dependences of the functional equation is cyclic). Accordingly, Mauch states that circularity is undesirable, if the PN represents a DP problem instance, because it can cause the DP solver to loop infinitely. Despite of this fact *they managed to model with gDPS* and solve by DP2PN2Solver the shortest path problem in cyclic graphs. Since the d-graph variants of Dijkstra and Bellman-Ford algorithms work in cyclic d-graphs too, the d-graph oriented software tool solves automatically the "circularity problem".

# 5. Conclusions

1. Developed as modeling tools for DP problems, the d-graph model moves DP problems through a concise intermediate representation to the field of graph theory, allowing optimal path/tree algorithms to find the optimal solutions.

2. The DP solver software efficiently implements the d-graph model to solve DP problems, even those with a "cyclic functional equation".

3. The high illustrative capability makes from the d-graph based DP solver software a useful didactic tool in teaching dynamic programming.

**Acknowledgment.** We are grateful to our colleague Ilyés László who was the first to observe the similarities between the Petri Nets and d- graphs.

# References

[1] Lew, A. (2002) A Petri net model for discrete dynamic programming. In: Proceedings of the 9th Bellman Continuum: International Workshop on Uncertain Systems and Soft Computing. Beijing, China, July 24–27, 16–21.

[2] Lew, A. and Mauch, H. (2004) Bellman nets: A Petri net model and tool for dynamic programming. In: Proceedings of Modelling, Computation and Optimization in Information Systems and Management Sciences (MCO).

[3] Mauch H, DP2PN2Solver: A flexible dynamic programming solver software tool, Control and Cybernetics, vol.35 (2006) No. 3.

[4] Zoltán Kátai, Dynamic programming and d-graphs, *Studia Universitatis Babes-Bolyai - Series Informatica*, LI (2006) 2, 41-52, ISSN: 1224869X.

[5] Kátai Zoltán, Dynamic programming as optimal path problem in wieghted digraphs, Acta Mathematica, Nyiregyháza, Hungary, 2008 (**??**)

[6] Zoltán Kátai, Ágnes Csíki, Automated dynamic programming, *Acta Universitatis Sapientiae, Informatica*, Vol. 1, No. 2, 149-164, 2009, ISSN: 1844-6086. (http://www.acta.sapientia.ro/)

[7] M. Sniedovich, Dijkstra's algorithm revisited: the dynamic programming connexion, Control and cybernetics, 35, 3 (2006) 599–620.

**Zoltán Kátai**
**Péter István Fülöp**
Sapientia University
Tirgu Mures / Corunca
Şoseaua Sighişoarei 1C
Romania
e-mail:
katai_zoltan@ms.sapientia.ro
fulop.peter.istvan@gmail.com