4-17-2006

Optimization Rules in DLV for the Bridge Crossing Problem

Sayan Ranu
Iowa State University

Prabhakar Balakrishnan
Indus Asuka

Gurpur M. Prabhu
Iowa State University, prabhu@iastate.edu

Optimization Rules in DLV for the Bridge Crossing Problem

Abstract
Disjunctive logic programming is a powerful tool in knowledge representation and commonsense reasoning. The first solid implementation of a DLP system is called DLV (Datalog with Vel). In this paper we offer three strategies to produce optimal solutions in DLV for the well-known Bridge Crossing Problem. These strategies are a piggyback strategy, a non-piggyback strategy, and a mixed strategy. An analysis to determine the number of time steps required for an optimal solution using these strategies is provided. We also characterize and prove the conditions under which a particular strategy should be used to obtain an optimal solution. These strategies are implemented in the form of optimization rules in a DLV program for the bridge crossing problem. Preliminary results indicate a drastic reduction in execution time when compared to other DLV programs for bridge crossing which do not incorporate these strategies. Our implementation uses a DLV Java wrapper, allowing us to embed disjunctive logic programs inside an object-oriented environment.

Keywords
Disjunctive logic programming, Datalog programs, bridge crossing problem, optimal strategies, DLV Java wrapper

Disciplines
Artificial Intelligence and Robotics

Technical Report 06-08 Department of Computer Science, ISU, April 17, 2006.

Optimization Rules in DLV for the Bridge Crossing Problem

Sayan Ranu[1], Prabhakar Balakrishnan[2] and G. M. Prabhu[1]

[1] Department of Computer Science, Iowa State University, Ames, IA 50011, USA
{sayanr, prabhu}@cs.iastate.edu
[2] Indusasuka, Chennai, India
{prabha}@indusasuka.com

Abstract. Disjunctive logic programming is a powerful tool in knowledge representation and commonsense reasoning. The first solid implementation of a DLP system is called DLV (Datalog with Vel). In this paper we offer three strategies to produce optimal solutions in DLV for the well-known Bridge Crossing Problem. These strategies are a piggyback strategy, a non-piggyback strategy, and a mixed strategy. An analysis to determine the number of time steps required for an optimal solution using these strategies is provided. We also characterize and prove the conditions under which a particular strategy should be used to obtain an optimal solution. These strategies are implemented in the form of optimization rules in a DLV program for the bridge crossing problem. Preliminary results indicate a drastic reduction in execution time when compared to other DLV programs for bridge crossing which do not incorporate these strategies. Our implementation uses a DLV Java wrapper, allowing us to embed disjunctive logic programs inside an object-oriented environment.

Keywords: Disjunctive logic programming, Datalog programs, bridge crossing problem, optimal strategies, DLV Java wrapper.

1 Introduction

Planning based on answer set programming is a powerful tool to express planning tasks in a declarative way. There are several declarative planning languages and formalisms available which allow encoding of complex planning problems using incomplete information, simultaneous actions, non-deterministic action effects, and ramifications. *K* is one such declarative planning language [1]. *K* supports transition between states of complete as well incomplete knowledge. Using *K*, one can solve problems of the form "find a sequence of actions that leads from a given initial state to a given goal state." K^c is an extension of language *K* by action costs [2]. K^c extends the notion of finding the optimal plan within a given number of steps by assigning costs to actions. This is particularly useful to find "cheapest" plans such that the total cost of all the actions is the lowest for that plan. Both these languages have been studied extensively in the literature [1, 2].

Disjunctive logic programming (DLP) with answer set semantics allows us to express very complex problems up to the $\sum^P_2$ complexity class [3, 4, 5]. DLV is a disjunctive logic programming engine, which is disjunctive datalog with answer set semantics. It is a highly portable program written in ISO C++ and available in binary form for many platforms [6]. Today, the DLV system is recognized to be the state-of-the-art implementation of DLP and is widely used for educational purposes in AI and database courses.

Our motivation for this paper arose from studying one of the popular problems which have been encoded in the languages DLV, *K*, and K^c, namely the Bridge Crossing Problem which is usually stated as follows.

Bridge Crossing Problem: *Four persons want to cross a river at night from side "here" to side "across" over a plank bridge, which can only hold up to two persons at a time. They have a lamp, which must be used when crossing. As it is pitch-dark and some planks are missing, someone must bring the lamp back to*

the others; no tricks (like throwing the lamp or halfway crosses, etc.) are allowed. The persons in the bridge crossing scenario need different times to cross the bridge. Walking in two implies moving at the slower rate of both. The goal is to get all four persons and the lamp from side "here" to side "across" in the smallest possible time.

Although the original problem is stated for four persons, it can be easily extended to more persons with the same crossing rules. We used existing encodings in DLV, K, and K^c for the Bridge Crossing problem and an executable engine available at http://www.dlvsystem.com/K. For the case of 4 persons crossing the bridge, the programs ran successfully and produced answers (or models as they are called) in a reasonable amount of time (seconds). But when we extended the problem to 5 persons, all these programs took an unduly large amount of time to produce a model. For the languages K, and K^c we could not find suitable programming environments to accurately measure execution times. However, for DLV, we found a DLV wrapper developed in [7]. The DLV wrapper is an object-oriented library that "wraps" up the DLV system in a Java program. It gives full control on DLV execution and allows the user to combine Java code with disjunctive logic programs. The Java wrapper acts as an interface between the DLV system and Java programs. This is particularly useful because a large number of software applications are developed using Java and the need to integrate these types of applications with logic-based systems are on the rise.

For the Bridge Crossing problem with 4 persons, a DLV program without optimization rules took 19 seconds to produce a solution. For 5 persons, the same program took 13 hours and 24 minutes. These execution times are on a Dell PowerEdge 4600 computer with a 2.2 GHZ Intel Xeon processor and 2 GB RAM running RedHat Linux. Although the search space increases exponentially with increase in problem size, we did not think that the execution times should be so high when going from a problem instance of size 4 to one of size 5. An analysis of the Bridge Crossing Problem gave rise to some provably optimal strategies which we also implemented in DLV. The execution times of our implementation using a DLV Java wrapper are drastically smaller – for example, a problem instance of size 5 is solved in 0.313 seconds and a problem instance of size 10 is solved in 2 seconds.

In Section 2 we describe three strategies for the Bridge Crossing Problem, namely a piggyback strategy, a non-piggyback strategy, and a mixed strategy, and show how to determine the strategy for the optimal solution. In Section 3 we provide, for each of the three strategies, an analysis of the optimal solution and the number of time steps required for optimality. Section 4 sketches the implementation details of a DLV program that implements these strategies. Execution times using our methods are reported for various values of problem size and crossing strategy. The DLV code for the Bridge Crossing Problem along with the implementation of our strategies as "optimization rules" is given in the appendix. For future work, we are examining how to design such optimal strategies for other problems and how to determine the extent to which such strategies can be automatically generated and included in the corresponding DLV programs.

2 Optimal Strategies for the Bridge Crossing Problem

An obvious strategy that first comes to mind is where the fastest person always keeps the lamp and leads all the others across the bridge. There are other strategies that are not as obvious.

Piggyback strategy: In this strategy, the fastest person keeps the lamp and escorts the others one at a time across the bridge.

Consider an example where the four persons are called A, B, C, and D. The times taken by them are 1 minute, 5 minutes, 8 minutes, and 10 minutes, respectively. In this example, the piggyback strategy works as follows:

```
Action 1: crossTogether (A, D) takes 10 minutes   // A and D cross together with the lamp
Action 2: cross (A) takes 1 minute                // A crosses back to side "here" with lamp
Action 3: crossTogether (A, C) takes 8 minutes
Action 4: cross (A) takes 1 minute
Action 5: crossTogether (A, B) takes 5 minutes
```

The total time taken is 25 minutes. The action crossTogether (x, y) is from side "here" to side "across" and either x or y must have the lamp. The action cross (x) is from side "across" to side "here" and x must have the lamp. Note that in this strategy there are no "takeLamp" actions because the fastest person always keeps the lamp. We assume in the initial state that the fastest person always has the lamp at side "here." However, the piggyback strategy does not always produce the optimal solution.

Non-piggyback strategy: The key idea behind this strategy is to allow the slowest two persons, who have not yet crossed, to travel together. However, we should allow this action only if one of the fastest two persons is already present on the other side of the bridge.

Consider an example where the times taken by the four persons are 1 minute, 2 minutes, 5 minutes, and 10 minutes, respectively. In this example, the non-piggyback strategy works as follows:

Action 1: crossTogether (A, B) takes 2 minutes
Action 2: cross (A) takes 1 minute
Action 3: takeLamp (D) // D needs the lamp to cross the bridge in the next step
Action 4: crossTogether (C, D) takes 10 minutes
Action 5: takeLamp (B) // takeLamp actions do not consume any time
Action 6: cross (B) takes 2 minutes
Action 7: crossTogether (A, B) takes 2 minutes

The total time taken is 17 minutes. We didn't allow the slowest two persons to travel together initially because none of the fastest two persons were present on side "across." Thus this action was only possible at time step 4 when person B, the second fastest, was present on the other side of the bridge.

The piggyback strategy for this example yields the following actions:

Action 1: crossTogether (A, D) takes 10 minutes
Action 2: cross (A) takes 1 minute
Action 3: crossTogether (A, C) takes 5 minutes
Action 4: cross (A) takes 1 minute
Action 5: crossTogether (A, B) takes 2 minutes

The total time taken is 19 minutes, which is clearly not optimal.

Mixed strategy: In some cases, to find the optimal solution, we need to apply both the non-piggyback and the piggyback strategies, depending on the crossing times.

Consider an example where we have six persons A, B, C, D, E, and F. The times taken by them are 1 minute, 5 minutes, 6 minutes, 7 minutes, 20 minutes, and 30 minutes, respectively. In this example, the mixed strategy works as follows:

Action 1: crossTogether (A, B) takes 5 minutes // Start out with non-piggyback strategy
Action 2: cross (A) takes 1 minute
Action 3: takeLamp (F)
Action 4: crossTogether (E, F) takes 30 minutes
Action 5: takeLamp (B)
Action 6: cross (B) takes 5 minutes
Action 7: takeLamp (A) // Switch to piggyback, A needs to have the lamp
Action 8: crossTogether (A, D) takes 7 minutes
Action 9: cross (A) takes 1 minute
Action 10: crossTogether (A, C) takes 6 minutes
Action 11: cross (A) takes 1 minute
Action 12: crossTogether (A, B) takes 5 minutes

The total time taken is 61 minutes. Note that we started out with the non-piggyback strategy till time step 6, and then switched to the piggyback strategy. This mixed strategy is used when both the non-piggyback and piggyback strategies do not produce optimal solutions. If we use the non-piggyback strategy throughout, the total time taken is 64 minutes, whereas for the piggyback strategy the total time is 72 minutes. We show below how to determine the correct strategy for finding an optimal solution. We also show when to switch from non-piggyback to piggyback in the mixed strategy.

Determining the Strategy for the Optimal Solution

Let us consider a general case where N persons have to cross the bridge. We denote the crossing time taken by person *i* as T_i. For the time being we make the assumption that the T_i's are distinct and the crossing times are sorted in increasing order, that is, $T_1 < T_2 < \ldots < T_N$. We also assume that initially the fastest person has the lamp.

We proceed inductively as follows. When N = 1, the optimal time to cross is T_1. When N = 2, the optimal time to cross is T_2. When N = 3, the optimal time to cross is $T_1 + T_2 + T_3$. In the inductive step when going from N down to N – 2, there are two cases to consider.

Case I (Piggyback)

Action	**Time**
crossTogether (1, N)	T_N
cross (1)	T_1
crossTogether (1, N-1)	T_{N-1}
cross (1)	T_1

Total time for Case I = $T_1 + T_N + T_1 + T_{N-1}$

Case II (Non-piggyback)

Action	**Time**
crossTogether (1, 2)	T_2
cross (2)	T_2
crossTogether (N-1, N)	T_N
cross (1)	T_1

Total time for Case II = $T_1 + T_N + 2* T_2$

By comparing the total times for the above cases, if $T_1 + T_{N-1}$ is smaller than or equal to $2* T_2$ then the piggyback strategy is optimal for the pair (N, N-1). Otherwise the non-piggyback strategy is optimal. For the case when both are equal, either strategy yields an optimal value (we use Case I without loss of generality).

Lemma 1. *If an optimal solution starts out with a piggyback strategy, then it will have to continue with it.*

Proof.
Since we start out with a piggyback strategy, it follows that $2* T_2$ is greater than or equal to $T_1 + T_{N-1}$ for the pair (N, N-1). After this pair is transported to the side "across," we have to compare $2* T_2$ with $T_1 + T_{N-3}$. Since the crossing times are assumed to be sorted in increasing order, T_{N-3} is smaller than T_{N-1}. Therefore $2 * T_2$ will be greater than $T_1 + T_{N-3}$. This relationship will continue to hold for all the remaining cases and thus the piggyback strategy yields the optimal solution. ■

Lemma 2. *In an optimal solution with the mixed strategy, once we switch from non-piggyback to piggyback, we have to continue with the piggyback strategy for the remaining persons who are on side "here" of the bridge.*

Proof.
Since the initial strategy is non-piggyback, it implies that $2* T_2$ is smaller than $T_1 + T_{N-1}$ for the pair (N, N-1). A switch from this strategy to a piggyback strategy can occur only when

$2* T_2 \geq T_1 + T_K$, where $K = N – (2m+1)$ for some integer $m > 0$, and $K > 2$.

Since the crossing times are assumed to be sorted in increasing order, this condition will continue to hold for smaller values of T_K. Thus the piggyback strategy, once it starts to kick in, will have to be continued in an optimal solution. ■

In the mixed strategy, the decision to switch or not should be made by comparing 2 * T_2 with the sum of the fastest and second slowest persons remaining on side "here." If 2 * T_2 is greater than or equal to this sum then we switch from non-piggyback to piggyback, else we continue with the non-piggyback strategy.

Corollary 3. *In the mixed strategy we always start out with a non-piggyback strategy.*

Proof. Follows directly from Lemmas 1 and 2. ■

3 Analysis of Optimal Solution and Number of Time Steps for Optimality

For the bridge crossing problem the minimum number of time steps required is equal to the number of actions needed to find an optimal solution. Time steps are required in DLV programs when you need to make some fluent true or false after a particular action or change in a fluent (a fluent characterizes a state and can be true, false, or unknown in a particular state). For example, the fluent *hasLamp(a)* becomes true after the *takeLamp(a)* action. Since simultaneous actions are not allowed in the bridge crossing problem, there can be at most one action per time step.

The number of time steps depends on the number of people and the strategy to find the optimal solution. We need to specify this number of time steps before the program is executed. The program will find an optimal solution only if a sufficient number of time steps are available. If we give a huge number of time steps, a solution will be found, but it will take much longer to execute since the search space increases exponentially. Therefore, to minimize the execution time we need to provide the smallest number of time steps required to find an optimal solution.

Before we execute the program, we already know the first factor, the number of people, on which the number of time steps depends. The second factor is the strategy. As is evident from the previous section, the piggyback strategy requires the least number of actions to transport *n* people. When compared to the non-piggyback strategy, it involves an equal number of cross and crossTogether actions, but it does not contain any takeLamp actions. Since the mixed strategy involves both piggyback and non-piggyback strategies, the total number of time steps required will lie in between the number of time steps required for these two strategies.

Number of Time Steps and Optimal Value for Piggyback Strategy

In the piggyback strategy, there are N – 1 **crossTogether** actions and N – 2 **cross** actions. Therefore the total number of actions (or time steps) required for the piggyback strategy with N people is

$$\text{Time Steps}_{Piggyback} = \mathbf{2N - 3}$$

$$\text{Optimal Value}_{Piggyback} = \mathbf{(N-2) * T_1 + \sum_{i=2}^{N} T_i}$$

Number of Time Steps and Optimal Value for Non-Piggyback Strategy

To find out the number of time steps required, we need to first determine the number of slowest and second slowest pairs from the formula $\llcorner (N - 2) / 2 \lrcorner$. Next, we need to calculate the number of time steps required to transport each of these pairs to the other side of the bridge and reduce the problem to that of (N – 2) people. This turns out to be six time steps (2 crossTogether, 2 cross, 2 takeLamp) for the first pair, and seven time steps (2 crossTogether, 2 cross, 3 takeLamp) for each of the remaining pairs. It is seven for each of the remaining pairs because it involves an extra takeLamp action for the fastest person, whereas for the first pair, the fastest person is already assumed to have the lamp. When all but 2 or 3 persons have been transported, there can be two situations:

1. Only the fastest and second fastest persons are present on the "here" side of the bridge. This will be true if N is even. In this case one more crossTogether action will solve the problem.

2. The fastest three persons are present on the "here" side of the bridge. This will be true if N is odd. In this case, three more crossTogether actions, and one takeLamp action will solve the problem.

Since we already know that the number of time steps is equal to the total number of actions, if we can derive a general formula for the total number of actions, we can determine the number of time steps needed to solve a particular problem using the non-piggyback strategy. From the above observations we derive the number of time steps as follows.

$$\begin{aligned}
\text{Time Steps}_{\textit{Non-piggyback}} &= 6 + 7 * (\lfloor (N-2)/2 \rfloor - 1) + (1 \text{ if N is even or 4 if N is odd}) \\
&= 6 + 7 * (\lfloor (N-2)/2 \rfloor - 1) + 1 + (0 \text{ if N is even or 3 if N is odd}) \\
&= 7 + 7 * (\lfloor (N-2)/2 \rfloor - 1) + (0 \text{ if N is even or 3 if N is odd}) \\
&= 7 * (1 + \lfloor (N-2)/2 \rfloor - 1) + (0 \text{ if N is even or 3 if N is odd}) \\
&= 7 * (\lfloor (N-2)/2 \rfloor) + (0 \text{ if N is even or 3 if N is odd}) \\
&= \mathbf{7 * (\lfloor (N-2)/2 \rfloor) + 3 * (N \bmod 2)}
\end{aligned}$$

To compute the optimal value we make the following observations. While transporting each of the pairs the following actions take place:

crossTogether (K, K-1) taking time T_K, where K = N, N – 2, N – 4,…, 5 or 4 depending on the value of N
crossTogether (1, 2) taking time T_2
cross (1) taking time T_1
cross (2) taking time T_2

Note that takeLamp actions have not been considered since they do not consume any time.

When all but 2 or 3 persons have been transported to the "across" side of the bridge, we need to add the time taken for transporting the last 2 or 3 persons on the "here" side, depending on the value of N.

- If N is even, then time taken at this stage will be T_2
- If N is odd, then time taken at this stage will be $T_1 + T_2 + T_3$

Thus the optimal value is determined as follows.

$$\begin{aligned}
\text{Optimal Value}_{\textit{Non-piggyback}} &= T_N + T_{N-2} + \ldots + (T_5 \text{ if N is odd, or } T_4 \text{ if N is even}) + T_1 * (\text{no. of pairs}) \\
&\quad + 2 * T_2 * (\text{no. of pairs}) + T_2 + \{(T_1 + T_3) \text{ if N is odd}\} \\
&= T_N + T_{N-2} + \ldots + \{T_4 \text{ if N even, } T_3 \text{ if N odd}\} + T_2 * (2 * \text{no. of pairs} + 1) + T_1 * (\text{no. of pairs}) \\
&\quad + \{T_1 \text{ if N is odd}\} \\
&= T_N + T_{N-2} + \ldots + \{T_4 \text{ or } T_3\} + T_2 * (2 * \lfloor (N-2)/2 \rfloor + 1) + T_1 * (\lfloor (N-2)/2 \rfloor) \\
&\quad + \{T_1 \text{ if N is odd}\} \\
&= \mathbf{T_N + T_{N-2} + \ldots + \{T_4 \text{ or } T_3\} + T_2 * (\lfloor N/2 \rfloor * 2 - 1) + T_1 * (\lfloor (N-1)/2 \rfloor)}
\end{aligned}$$

Number of Time Steps and Optimal Value for Mixed Strategy

Since this involves both piggyback and non-piggyback strategies, we know that the number of time steps required will be between (2N – 3) and 7 * (⌊ (N – 2) /2 ⌋) + 3 * (N mod 2). Recall that the piggyback strategy requires the least number of time steps and the non-piggyback strategy requires the most number of time steps. The number of time steps for mixed strategy will thus depend on the number of pairs we transport using the non-piggyback strategy. If M is the number of pairs transported using the non-piggyback strategy then the number of time steps will be

$$\text{Time Steps}_{\textit{Mixed}} = 7*M + 3 * ((2*M) \bmod 2) + 2*(N - 2*M) - 3$$

$$= \mathbf{7*M + 2 * (N - 2*M) - 3} \qquad \text{since } 2*M \text{ is even}$$

We now need to determine M to find the exact value for the above formula. Recall from Corollary 3, that in a mixed strategy we always start out with the non-piggyback strategy. We switch to the piggyback strategy for some K, such that $2*T_2$ is greater than or equal to $T_1 + T_K$, where K = N – 1, N – 3, N – 5, …, 4 or 3, depending on the value of N. We also know from Lemma 2 that once the switch occurs, the optimal strategy remains piggyback for the rest of the persons on the "here" side of the bridge. Therefore we can say that,

$\mathbf{M = (\llcorner (N - K) / 2) \lrcorner)}$ where $2 < K < (N - 2)$.

Thus the total number of time steps can be calculated by determining where the switch to the piggyback strategy occurs.

To compute the optimal value, we divide the persons into two sets.

- The first set contains pairs transported using the non-piggyback strategy. Assume the number of pairs in Set 1 is M (we exclude the fastest and second fastest persons since they are not considered one of these pairs, although they do play a role in taking the lamp across). Therefore total number of people transported using the non-piggyback strategy is 2*M. These people will be persons N to (N – 2(M – 1) – 1). Thus pairs will be formed between N and (N – 1), (N – 2) and (N – 3), and so on till (N – 2 (M – 1)) and (N – 2 (M – 1) – 1). The crossing time for each pair is the larger of the two times of the pair.

- Set 2 contains persons transported using the piggyback strategy, which consists of person (N – 2 (M – 1) – 2) to person 2 (we exclude the fastest, since he escorts each person in this set).

Optimal value for Set 1 = $(T_N + T_{N-2} + \ldots + T_{(N-2(M-1))}) + (T_2 * (2*M) + T_1 * M)$

The first portion in parentheses contains the sum of the times taken by each of the pairs. The second portion comes from the fact that there are two actions taking time T_2 and one action taking time T_1 during the transportation of each pair from side "here" to side "across."

Optimal Value for Set 2 = $((N - 2 (M - 1) - 2) - 2) * T_1 + \sum_{i=2}^{(N-2(M-1) - 2)} T_i$

Therefore, the optimal value to transport all the people across the bridge turns out to be

$$\text{OptimalValue}_{Mixed} = (T_N + T_{N-2} + \ldots + T_{(N-2(M-1))}) + T_2 * (2*M) + T_1*M + T_1 * (N - 2(M+1)) + \sum_{i=2}^{N-2M} T_i$$

$$= \mathbf{T_N + T_{N-2} + \ldots + T_{(N-2(M-1))} + T_2 * (2*M) + T_1 * (N - M - 2) + \sum_{i=2}^{N-2M} T_i}$$

4 Execution through DLV Java Wrapper

Implementation of the Java Program

We execute the plain DLV program for the bridge crossing problem using the DLV Java Wrapper described in [7]. The DLV Java Wrapper is an object-oriented library that allows the user to embed the DLV system into a Java program. The Java Wrapper acts as an interface between the DLV system and the Java program. By using a suitable hierarchy of Java classes, the DLV Java Wrapper combines Java code with DLV programs.

We have extended the Java wrapper in [7] for our implementation. This gives us complete control over the DLV execution. Our program has the following advantages:

- The Java program interacts with the user and adds rules and information (called "background knowledge") to the DLV program based on the inputs provided. This allows the program to automatically compute the number of time steps required, based on the walking times provided by the user. Moreover, changing the DLV code manually is very error prone, and it's extremely hard to debug. The Java program allows us to automate this process.
- The Java program is able to record the execution time for a particular instance.
- The Java program stores and updates results in a file.

Our Java implementation consists of three steps:

1. Set up input and add DLV rules
2. Set up and run DLV from the Java Wrapper (extension of [7])
3. Handle output

In Step 1, we set up the program to interact with the user. In the DLV program, the background knowledge is not constant. The number of persons, their walking times, and the "time" and "next" clauses vary. The Java program takes the number of persons, and the walking times as inputs from the user. Then it creates the DLV rules accordingly and adds them to the DLV program. The user can either ask the Java program to compute the number of time steps automatically or enter it manually. The DLV Java Wrapper library provides methods which allow us to create rules in the form a "String" which can then be added to the DLV program. The Java program also requires the user to input the number of models to compute. This ensures that the DLV engine does not waste time in computing more models than what is needed. Thus even if more models exist in the search space, the DLV engine outputs only the number of models required by the user. If the search space contains a fewer number of models than required by the user, the DLV engine outputs all of them.

In Step 2, we set up the invocation parameters to run the DLV system. This includes creating a *DLVHandler* object to set up invocation parameters, and run DLV process instances. The *DLVHandler* gives us full control on DLV execution. The *DLVHandler* object calls DLV, feeding input and retrieving output. As soon as DLV outputs a new model, the *DLVHandler* object parses it into a readable format and outputs it. The output is displayed on the screen. We also set up a timer before executing the DLV program. The timer stops after the DLV engine stops executing. The execution time is displayed on the screen below the DLV output. This step is primarily an extension of Java code from the basic DLV Java wrapper found in [7].

In Step 3, we process the output and record it in a file named "bridge.out." Each time the Java program is run, the following entries are added to the file:

- The number of time steps in this solution
- The number of persons
- The walking time of each person
- Model found or not
- The execution time

All this information is appended to the "bridge.out" file, keeping the previous records intact.

Executing the Java Program

The program can be compiled and run using the "javac" and "java" commands. The program is named "BCP.java." We assume that the DLV engine is stored in the same directory. The program takes three inputs when called through the command line using "java."

1. The first input is the name or the path of the DLV file to be executed. If the DLV file is present in the same directory where BCP is then only the name of the file will do. Otherwise, the path to the file needs to be entered.

2. The second input is the number of models the user wants. If the search space contains a fewer number of models than required by the user, the DLV engine outputs all of them.

3. The third input is the number of time steps. It can be either "auto" or some integer. The input "auto" makes the program calculate the time steps required for the optimal solution automatically, using the formulas developed in Section 3. This saves time for the user who doesn't have to calculate it manually. The program prints the number of time steps before producing the solution. The "auto" parameter always produces the least number of time steps in which an optimal solution is possible. Thus if the user enters any time step smaller than the one produced by "auto," the DLV engine won't be able to find a solution and output "I cannot find a model." On the other hand, any time step larger than the one produced by "auto" will give a solution, but will take more time to execute.

A typical command to run the program looks like

```
java BCP bridge.dlv 1 auto
```

Here "bridge.dlv" is the name of the DLV program that the DLV engine will execute. The number of models specified is "1." Here only one model will be printed, if a model exists. The third argument "auto" makes the Java program calculate the time steps required for the optimal solution automatically.

After this, the program asks for the number of people. Once this value is entered, the program asks for the walking times of each of these persons. We name the persons alphabetically starting from "a." We assume that the walking times are entered in ascending order and are distinct. Once the walking times are entered, the Java program creates the DLV rules from these inputs and adds them to the DLV program. Note that we don't change the DLV file; we simple create a DLV program which contains the rules from both the files as well as the ones created by the Java program. Then the DLV engine to execute the DLV program is called from within the wrapper.

The DLV Program (Code provided in Appendix)

The DLV program is given in the appendix and consists of two parts. The first part is the general DLV program which will work on its own. The second part contains the optimization rules which drastically reduce the execution time. To compare the times of the DLV program with and without the optimization rules you can run the first part separately with identical inputs. The table in the "Execution Times" section gives a comparison of execution times with and without optimization rules.

Optimization of Bridge Crossing Problem

The goal in optimizing is to eliminate successful models (optimal or not optimal) from the search space but ensure that there is at least one optimal model left in the search space. The execution time depends on the number of models in the search space. If DLV has a fewer number of models in its "search set" for the optimal solution, then it will take less time to come up with a model. Thus all the optimization rules eliminate models which are either not optimal, or are a variation of another optimal model. The goal is to reduce the search space to as few models as possible but that still contain an optimal solution. In our current implementation, the optimization rules accomplish this reduction. While some of the rules may appear unnecessary, they do reduce the search space and hence the execution time.

Our optimization rules have been divided into three major sections.

1. General Optimization Rules
2. Simplification of Optimization Rules
3. Implementation of Strategies

The first section contains general optimization rules [8] which are true regardless of the strategy used. For example, there can never be a *crossTogether* action from side "across" to "here" and a *cross* action from side "here" to "across" in an optimal solution. These types of rules have been implemented in our program.

The second section contains optimization rules which are used later on in the piggyback and non-piggyback strategies. We calculate the fastest, second fastest, slowest, and second slowest persons in this section. Although these calculations are not necessary because the times are assumed to be in increasing order, we provide them to make the code more general. The strategy to be followed is also calculated in this section. If the optimal strategy is mixed, then the switch from non-piggyback to piggyback occurs in this section.

The third section contains rules which implement the piggyback and non-piggyback strategies. Since the mixed strategy is a combination of the both, these rules are sufficient to implement the mixed strategy. The DLV code contains detailed comments which explain the reasons behind each of the optimization rules.

Execution Times

The times reported are for the DLV code with and without optimization when run on a Dell PowerEdge 4600 computer with a 2.2 GHZ Intel Xeon processor and 2 GB RAM running RedHat Linux.

Number of People	Walking Times	Number of Time Steps	Optimal Strategy	Execution Time		
				DLV with optimizations	DLV	K^C
4	{1,5,6,7}	auto (5)	Piggyback	0.148 secs	1.09 secs	0.5 secs
4	{1,2,5,10}	auto (7)	Non-piggyback	0.21 secs	19.38 secs	2 secs
5	{1,8,9,12,13}	auto (7)	Piggyback	0.226 secs	4 mins 14 secs	11 secs
5	{1,8,9,20,30}	auto (10)	Non-piggyback	0.313 secs	13 hrs 24 mins	> 2 hours

Number of People	Walking Times	Strategy	Number of Time Steps	Optimal Solution	Execution Time (in seconds)
4	{1,5,6,7}	Piggyback	auto (5)	20	0.148
4	{1,2,5,10}	Non-Piggyback	auto (7)	17	0.21
5	{1,8,9,12,13}	Piggyback	auto (7)	45	0.226
5	{1,8,9,12,13}	Piggyback	10	45	0.3
5	{1,8,9,20,30}	Non-piggyback	7	No model found	0.172
5	{1,8,9,20,30}	Non-piggyback	auto (10)	65	0.313
6	{1,2,5,10,11,12}	Non-piggyback	auto (14)	34	0.638
6	{1,5,7,10,11,12}	Mixed	auto (12)	47	0.58
7	{1,7,8,9,10,12,15}	Piggyback	auto (11)	65	0.621
7	{1,3,7,8,10,12,15}	Non-piggyback	auto (17)	50	1.184
8	{1,6,7,10,11,12,13,14}	Mixed	auto (16)	77	5.624
8	{1,7,8,9,10,11,12,13}	Piggyback	auto (13)	76	0.897
9	{1,2,6,7,10,12,13,14,15}	Non-piggyback	auto (24)	62	3.403
9	{1,5,6,7,10,14,16,18,20}	Mixed	auto (21)	89	3.429
10	{1,10,11,12,..,17,18}	Piggyback	auto (17)	134	2.149
10	{1,7,8,9,10,11,14,15,18,20}	Mixed	auto (23)	114	10.837
10	{1,2,5,10,11,…,15,16}	Non-piggyback	auto (28)	74	4.92
15	{1,15,16,..,27,28}	Piggyback	auto (27)	301	17.825
15	{1,2,5,10,11,...,20,21}	Non-piggyback	auto (45)	134	50.84
20	{1,20,21…,37,38}	Piggyback	auto (37)	534	1 min 32 secs
20	{1,2,5,10,11,…,25,26}	Non-piggyback	auto (63)	189	4 mins 51 secs
20	{1,5,6,7,8,..,22,23}	Mixed	auto (58)	235	5 mins 30 secs

5 Conclusions

We presented three strategies to produce optimal solutions for the Bridge Crossing problem. These strategies were analyzed to determine the number of time steps required for optimality. Closed-form expressions for the optimal value were provided for each strategy. These strategies were implemented in the form of optimization rules in a DLV program for the problem. Our implementation used a DLV Java wrapper that gave us more control over the execution environment and enabled us to accurately measure the execution times for a number of different inputs. Our results demonstrate a drastic reduction in execution times when compared to DLV programs which do not use these strategies. For future work we are examining how to design such optimal strategies for other problems and how to determine the extent to which such strategies can be automatically generated and embedded in the corresponding DLV programs.

References

[1] Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A., Planning under Incomplete Knowledge. In: *Proc. of CL 2000*. Lecture notes in Computer Science, Vol. 1861. Springer-Verlag, (2000) 807-821.

[2] Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A., Answer Set Planning Under Action Costs. *Journal of Artificial Intelligence* 19 (2003) 25-71.

[3] Gelfond, M. and Lifschitz, V., Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365-385, 1991.

[4] Lobo, J., Minker, J., and Rajasekar, A., *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[5] Leone, N., Rullo, P., and Scarello, F., Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation,* 135(2): 69-112, 1997.

[6] Faber, W. and Pfeifer, G., DLV Homepage since 1996, http://www.dlvsystem.com/.

[7] Ricca, F., and Leone, N., The DLV Wrapper Web Site, http://www.mat.unical.it/wrapper/index.html

[8] Rote, G., Crossing the Bridge at Night, http://www.inf.fu-berlin.de/inst/ag-ti/publications/rote.en.html

APPENDIX

The DLV Code for Bridge Crossing

```
#maxint=100.
max(A,B,A):- walk(_,A), walk(_,B), A>=B.
max(A,B,B):- walk(_,A), walk(_,B), A<B.
side(here). side(across).
otherside(X,Y):- side(X), side(Y), X!=Y.

at(X,here,0) :- person(X).
hasLamp(a,0).

crossTogether(X,Y,T0) v -crossTogether(X,Y,T0) :- hasLamp(X,T0), at(X,S,T0),
side(S), at(Y,S,T0), person(X), person(Y), next(T0,T1), X != Y, left(T0).
crossTogether(X,Y,T0) v -crossTogether(X,Y,T0) :- hasLamp(Y,T0), at(X,S,T0),
at(Y,S,T0),side(S), person(X), person(Y), next(T0,T1), X != Y, left(T0).

cross(X,T0) v -cross(X,T0) :- hasLamp(X,T0), person(X), next(T0,T1), left(T0).

takeLamp(X,T0) v -takeLamp(X,T0) :- person(X), person(Y), hasLamp(Y,T0), at(X,S,T0),
at(Y,S,T0), X!=Y, next(T0,T1),left(T0).

at(X,S1,T1) :- crossTogether(X,Y,T0), at(X,S,T0), otherside(S,S1), person(X),
person(Y), next(T0,T1), side(S1), side(S).
at(Y,S1,T1) :- crossTogether(X,Y,T0), at(Y,S,T0), otherside(S,S1), person(X),
person(Y), next(T0,T1), side(S1), side(S).
-at(X,S,T1) :- crossTogether(X,Y,T0), at(X,S,T0), person(X), person(Y), next(T0,T1),
side(S).
-at(Y,S,T1) :- crossTogether(X,Y,T0), at(Y,S,T0), person(X), person(Y), next(T0,T1),
side(S).

at(X,S1,T1) :- cross(X,T0), at(X,S,T0), otherside(S,S1), person(X), next(T0,T1),
side(S1), side(S).
-at(X,S,T1) :- cross(X,T0), at(X,S,T0), person(X), next(T0,T1), side(S).

hasLamp(X,T1) :- takeLamp(X,T0), person(X), next(T0,T1).
-hasLamp(X,T1) :- takeLamp(Y,T0), hasLamp(X,T0), person(Y), person(X), next(T0,T1).


at(X,S,T1) :- not -at(X,S,T1), at(X,S,T0), person(X), next(T0,T1), side(S).
hasLamp(X,T1) :- not -hasLamp(X,T1), hasLamp(X,T0), person(X), next(T0,T1).

:- cross(X1,T0), cross(X2,T0), X1 != X2.
:- cross(X1,T0), crossTogether(X2,X3,T0).
:- cross(X1,T0), takeLamp(X2,T0).
:- crossTogether(X1,X2,T0), crossTogether(X3,X4,T0), X1 != X3.
:- crossTogether(X1,X2,T0), crossTogether(X3,X4,T0), X2 != X4.
:- crossTogether(X1,X2,T0), takeLamp(X3,T0).
:- takeLamp(X1,T0), takeLamp(X2,T0), X1 != X2.

costs_cross(X,T,WX) :- cross(X,T), walk(X,WX).
:~ costs_cross(X,T,WX). [WX:]

costs_crossTogether(X,Y,T,Smax) :- crossTogether(X,Y,T), walk(X,WX), walk(Y,WY),
max(WX,WY,Smax).
:~ costs_crossTogether(X,Y,T,Smax). [Smax:]


left(T0) :- person(X), at(X,here,T0), time(T0).


goal :- not left(T0), time(T0).
:- not goal.
```

Optimization Rules

```
% ============================ General optimization rules ==========================================

%In an optimal solution a crossTogether action cannot be initiated from side "across"
:- crossTogether(X,Y,T0), person(X), person(Y), at(X,across,T0), time(T0).
:- crossTogether(X,Y,T0), person(X), person(Y), at(Y,across,T0), time(T0).

%In an optimal solution a cross action cannot be initiated from side "here" for a single person
:- cross(X,T0), at(X,here,T0), person(X), time(T0).

%In an optimal solution the fastest person should cross to side "here"
:- -cross(X,T0), at(X,across,T0), fastest(X), time(T0).

%In an optimal solution there can never be two consecutive takeLamp actions.
:- takeLamp(X,T0), takeLamp(Y,T1), person(X), person(Y), next(T0,T1).

%============================Simplification of optimization rules================================

% This rule forces crossTogether to treat the arguments as an ordered pair than a set.
% For example, crossTogether(a,b) is same as crossTogether(b,a), but DLV treats them as different models.
:-  walk(X,WX), walk(Y,WY), WX<WY, crossTogether(Y,X,T0), time(T0).

% Determination of fastest, second fastest, slowest, and second slowest
% The slowest and second slowest persons are determined among people who are on the "here" side
% The fastest and second fastest persons are same throughout the execution
% We assume each person has a distinct walking time

% X is fastest if there doesn't exist a person who takes less time
fastest(X) :- person(X), walk(X,WX), not existsLower(WX).
existsLower(WX) :- walk(X,WX), walk(Y,WY), WY < WX.

% X is second fastest if there doesn't exist a person who takes less time and not the fastest
secFastest(X) :- person(X), walk(X,WX), not secExistsLower(WX), not fastest(X).
secExistsLower(WX) :- walk(X,WX), walk(Y,WY), not fastest(Y), WY < WX.

% X is slowest if there doesn't exist a person who takes more time and is on the "here" side
slowest(X,T) :- person(X), walk(X,WX), not existsLarger(WX,T), at(X,here,T), time(T).
existsLarger(WX,T) :- walk(X,WX), walk(Y,WY), WY > WX, at(X,here,T), at(Y,here,T), time(T).

% X is slowest if there doesn't exist a person who takes more time, is on the "here" side and not the slowest
secSlowest(X,T) :- person(X), walk(X,WX), not secExistsLarger(WX,T), not slowest(X,T), at(X,here,T), time(T).
secExistsLarger(WX,T) :- walk(X,WX), walk(Y,WY), not slowest(Y,T), WY > WX, at(X,here,T), at(Y,here,T),
time(T).

%====Rules to determine strategy: piggyback or non-piggyback=====

% Rule: If 2*T2 >= T1 + T(n-1) then switch to piggyback strategy.

piggyBack(T0) :- not non_piggyBack, not checked(T0), secFastest(X), walk(X,WX), fastest(Y), walk(Y,WY),
secSlowest(Z,T0), walk(Z,WZ), time(T0), K = WY + WZ, I = WX + WX, I = K+J, J>=0, hasLamp(X,T0), at(X,here,T0),
at(X,here,T0), at(Y,here,T0), Z!=X.

piggyBack(0) :- not non_piggyBack, secFastest(X), walk(X,WX), fastest(Y), walk(Y,WY), secSlowest(Z,0),
walk(Z,WZ), time(0), K = WY + WZ, I = WX + WX, I = K+J, J>=0, hasLamp(Y,0), Z!=X.

% Once you have switched to piggyback continue with this strategy, because future T(n-1)'s will be smaller
piggyBack(T1):- piggyBack(T0), next(T0,T1).
checked(T1) :- piggyBack(T0), time(T1), time(T0),T1>T0.

num(X,WX) :- walk(X,WX), WX >= 0.
number(N) :- #count{X : num(X,WX)} = N.

non_piggyBack:- person(a), walk(a,WA), person(b), walk(b,WB), person(c), walk(c,WC),
               K = WA + WC, I = WB + WB, K= I+J, J>0, number(N), N=2*M, M>0.

non_piggyBack:- person(a), walk(a,WA), person(b), walk(b,WB), person(d), walk(d,WD),
               K = WA + WD, I = WB + WB, K= I+J, J>0, number(N), N=P + 1, P=2*M, M>0.

nonpiggyBack(T0) :- not non_piggyBack, not piggyBack(T0), time(T0).
nonpiggyBack(T0) :- non_piggyBack, time(T0).
```

```
%======================== Rules or Constraints for Piggyback Strategy ================================

% Ensures that only the fastest person is allowed to cross to the "here" side
:- piggyBack(T0), cross(X,T0), not fastest(X),at(X,across, T0), time(T0).

% No one is allowed to take the lamp from the fastest person
:- piggyBack(T0), hasLamp(Y,T0), fastest(Y), takeLamp(X,T0), time(T0).

% Ensures that fastest and slowest will always crossTogether when fastest is on side "here"
:- piggyBack(T0), -crossTogether(X,Y,T0), fastest(X), slowest(Y,T0),at(X,here, T0),at(Y,here, T0), time(T0).

% It's not possible to have a crossTogether action where the faster person is not the fastest
:- piggyBack(T0), crossTogether(X,Y,T0), not fastest(X),at(X,here, T0),at(Y,here, T0), time(T0).

% It's not possible to have a crossTogether action where the slower person is not the slowest
:- piggyBack(T0), crossTogether(X,Y,T0), not slowest(Y,T0), at(X,here, T0),at(Y,here, T0),time(T0).

%======================== Rules or Constraints for Non-Piggyback Strategy ================================

% Ensures that the slowest and the second slowest persons will always crossTogether when the second fastest
% person is on the "across" side
:- -crossTogether(B,A,T0), slowest(A,T0), secSlowest(B,T0), at(X,across,T0), secFastest(X), time(T0),
nonpiggyBack(T0).

% Ensures that the fastest and the second fastest persons will always crossTogether when both are on "here"
% side
:- -crossTogether(X,Y,T0), at(Y,here,T0), secFastest(Y), at(X,here,T0), fastest(X), time(T0), nonpiggyBack(T0).

% Ensures that if the second fastest person is on the "across" side he always brings the lamp back to the
% "here" side when the fastest is on "here" side
:- -cross(X,T0), at(X,across,T0), fastest(Y), at(Y,here,T0), secFastest(X), time(T0), nonpiggyBack(T0).

% Ensures that if a person is not the fastest or second fastest, then he is not allowed to do a cross action
:- cross(X,T0), not fastest(X), not secFastest(X), time(T0), nonpiggyBack(T0).

% Ensures that when the fastest and second fastest person are on the "here" side no other persons are allowed
:- crossTogether(B,A,T0),  at(Y,here,T0), fastest(X), secFastest(Y), B!=X, time(T0), nonpiggyBack(T0).
:- crossTogether(B,A,T0),  at(Y,here,T0), secFastest(Y), A!=Y, time(T0), nonpiggyBack(T0).

% Ensures that if you are not the second slowest-slowest, or fastest-second fastest then you are not allowed
% to do a crossTogether action
:- crossTogether(B,A,T0), not slowest(A,T0), not secFastest(A), not secSlowest(B,T0), not fastest(B), time(T0),
nonpiggyBack(T0).

% Ensures that when the second fastest person is on the "across" side no person other than the slowest and
% second slowest persons are allowed to take part in a crossTogether action
:- crossTogether(B,A,T0), at(Y,across,T0), secFastest(Y), slowest(Z,T0), A!=Z, time(T0), nonpiggyBack(T0).
:- crossTogether(B,A,T0), at(Y,across,T0), secFastest(Y), secSlowest(Z,T0), B!=Z, time(T0), nonpiggyBack(T0).

% Ensures no one can take the lamp other than the slowest person on the "here" side
:- takeLamp(X,T0), not slowest(X,T0),  at(X,here,T0), time(T0), nonpiggyBack(T0).

% Ensures no one can take the lamp other than the fastest and second fastest persons on the "across" side
:- takeLamp(X,T0), not secFastest(X), not fastest(X), at(X,across,T0), time(T0), nonpiggyBack(T0).

% Ensures no one can take the lamp when the second fastest person is on the "here" side
:- takeLamp(Z,T0), secFastest(X), at(X,here,T0), at(Z,here,T0), time(T0), nonpiggyBack(T0).

% Ensures that the slowest person always takes the lamp when the second fastest person is on the "across" side
% and the fastest person on the "here" side
:- -takeLamp(Z,T0), slowest(Z,T0), secFastest(Y), fastest(X),  at(X,here,T0), at(Y,across,T0), hasLamp(X,T0),
not secSlowest(X,T0), time(T0), nonpiggyBack(T0).

% Ensures that the second fastest person always takes the lamp when the second fastest person is on the
% "across" side, the fastest person on the "here" side, and someone on the "across" side has the lamp
:- -takeLamp(Y,T0), secFastest(Y), fastest(X),  at(X,here,T0), at(Y,across,T0), time(T0), nonpiggyBack(T0).

% Ensures that the fastest person always takes the lamp when he is on the "across" side
:- -takeLamp(X,T0), secFastest(Y), fastest(X),  at(X,across,T0), at(Y,across,T0), time(T0), nonpiggyBack(T0).
```