IMPERIAL COLLEGE LONDON

Department of Computing

Approximate Dynamic Programming: Playing Tetris and Routing Under Uncertainty

Michael Hadjiyiannis

Supervised by
Dr Daniel Kuhn

06/16/2009

MEng Final Year Project

Abstract

Tetris can be formulated as a stochastic control problem. However, the so called curse of dimensionality gives rise to prohibitive computational requirements that render infeasible the exact solution of the arising optimization model. In this project, we investigate a modern approximate dynamic programming technique called the BRLP method, that has been shown to defeat the curse of dimensionality and provide a near-optimal policy for playing Tetris. The technique reduces the computational load by intelligently sampling states through simulation, which are then used to formulate a specific linear optimization problem. The BRLP method is designed to iteratively refine the solution until the optimal policy is derived. We use the method to derive a near-optimal policy for Tetris. We experiment with a number of extensions for the method, in order to see if we can improve the solution. The proposed variations aim to expose why the method seizes to work after a certain number of iterations. In a second step, we look at a routing problem. The goal is to transmit a maximum amount of data through a network whose links have uncertain bandwidths. We are allowed to measure only a limited number of links per unit time, and the uncertainty about the bandwidths increases with the time since the last measurement. We formulate the problem as a stochastic control problem, and we use the BRLP method to derive a clever policy for choosing which links to measure. We evaluate the policy through simulations of various networks, and we compare it against a round robin policy.

Acknowledgements

I would like to thank my family for their support, especially my parents. Without their encouragement and UUU-type funding (Unbounded, Unconditional and Unconstrained) it would have not been possible to finish this report. I would also like to thank my supervisor, Dr Daniel Kuhn for his constant support and motivation throughout the entire year. I would like to acknowledge my second supervisor Professor Berç Rustem for his constructive feedback and support.

Contents

CHAPTER 1

Introduction

Optimal Routing is one of the central problems that computer scientists are called to solve. This is evident from the vast number of routing problem formulations, ranging from the Traveling Salesman Problem (TSP) to the Canadian Traveler problem, the Vehicle Routing Problem (VRP) and the numerous variations of those, such as the Vehicle Routing Problem with Pickup and Delivery(VRPPD), the Vehicle Routing Problem with LIFO,and the Traveling Salesman Problem with Stochastic Customers (TSPSC). The rationale behind the rigorous study of routing problems is attributable to their application on real life situations. Of greater interest, are the routing problems that incorporate uncertain parameters, called stochastic routing problems, due to their capability to capture the nature of real life problems, such as uncertain demand at a node, or uncertain cost for traversing an edge.

In its most general form, the stochastic routing problem describes the class of shortest path problems where the weight of an edge is uncertain. Because of its structure,this general problem, and in fact most of the stochastic routing problems, can be formulated as Markov Decision Processes. This in turn implies that dynamic programming can be used to solve these problems, as demonstrated in [1][2],[Vol 1. Chap 2][3],[4].

Dynamic programming attempts to calculate the exact cost-to-go function for each state, i.e. the expected cost for reaching the destination, given that we are at the current node and we will use the optimal path to reach the destination. This calculation is done by solving Bellman's equation. As a result, the stochastic routing problem reduces to simply choosing as the next edge to traverse, the one with the lowest cost-to-go value. However, exact Dynamic Programming suffers from the 'curse of dimensionality' and thus cannot be used for problems with large state spaces. This effectively renders DP inapplicable for optimizing real life problems, where thousands of optimization parameters exist.

To overcome the 'curse of dimensionality' a separate branch of Dynamic Programming has been developed, called Approximate Dynamic Programming.

Using techniques such as state augmentation,temporal differences,policy projections, simulation and sampling, approximate dynamic programming reduces the stochastic problem into the easier problem of finding a good cost-to-go approximation. Several methods have been designed to come up with good approximations. They have been used with great success in various situations suffering from the curse of dimensionality. A popular example is the Tetris game. There are approximately 10^{61} states in a game of Tetris. Designing a program that will play Tetris intelligently has always attracted a lot of attention from the field [4] [5] [6][7].

Recently, a state of the art technique in Approximate Dynamic Programming called the BRLP method, was successfully used to solve the Tetris problem with impressive results. The method resulted in a cost-to-go approximation for the Tetris game that results in a controller which scores an average of 4300 points(1 point per line) in a game [6]. The method utilizes a set of mechanisms to defeat the 'curse of dimensionality'. State augmentation , where only a set of important features of a state are considered, simplifies the stochastic problem to that of finding good cost-to-go approximation functions, i.e. functions that assign a value to different features of a particular state, and then calculating weightings for summing up these values. The weighted sum serves as an approximation to the true cost-to-go. For the weight calculation , BRLP employs an ingenious constraint sampling mechanism, where large samples of states are collected through simulation of the Tetris game. The samples are then used to define a Linear Problem, which is specially designed to calculate the optimal weights for that sample. The BRLP method iteratively refines the weights, using a process called Bootstrapping. By using the derived weights, the method improves the sampling mechanism and generates a new sample set which is then used to derive an even better set of weights. The method repeats this process several times, until the performance of the solution no longer improves.

In this project, we aim to use the BRLP method in a specific routing problem, which also suffers from the 'curse of dimensionality'. Our objective is to repeatedly transmit a maximum amount of data through a network whose links have uncertain bandwidths. The transmission must be robust, in that at no point in time must we transmit through a link at a higher bandwidth that what the link cope with. Some sort of bandwidth measurement is allowed but only on a limited number of links per unit time, right before transmission. The uncertainty about the bandwidths increases with the time since the last measurement. By measuring the right links each time, we reduce our uncertainty about the links and establish the links' present bandwidths. The right measurements and a bit of luck might result in us being able to transmit at a higher bandwidth. The problem is in deciding which links to measure, based only on the information that is available to us, such as the time when the link was last measured.

1.1 Contributions

- The BRLP method utilizes several techniques to overcome different aspects of the 'curse of dimensionality'. The method refines the solution iteratively, and in theory, the method should keep improving the solution until the optimal cost-to-go approximation is derived. However, the method stops improving the solution after 4 to 5 iterations. We investigate the method in more detail, and suggest extensions that might either explain or overcome this limitation (Chapter 3).

- We create a Tetris simulator and apply the method in the Tetris problem, in an attempt to reproduce the results reported in [6], where the method resulted in a policy that scored 4300 points on average (Chapter 4). We experiment with different settings for the BRLP method and we improve the performance of the resulting policy by 2000 points, up to a level of 6000 points on average. We investigate the proposed extensions on the Tetris problem, in order to measure the impact they have on the solution. From the results, we suggest a reason why the BRLP method fails after a certain number of iterations. We use a separate method (Least Square) to solve the Tetris, in order to support our claims. The Least Square method results in a policy that scores 11000 lines on average(Chapter 6).

- We formulate the routing problem as a Markov Decision Problem so that it can be solved by the BRLP. We design a set of functions that are used in the cost-to-go approximation. We devise the performance objectives that we want the solution of the problem to achieve. We modify the BRLP method so that it can optimize the problem. We implement a network simulator that we use for the BRLP method, as well as a controller that can control the network (Chapter 5).

- We experiment with the routing problem using the BRLP method. We identify the best settings for the BRLP method, in order to apply it on the routing problem. Through various simulations, we demonstrate that the BRLP method can be successfully used to solve the routing problem, and that an intelligent controller can be designed to operate it(Chapter 6).

CHAPTER 2

Background

This chapter aims to introduce the notions that are necessary for understanding approximate dynamic programming, as well as the kind of problems it was designed to solve.

2.1 Markov Decision Processes

Named after Russian Mathematician Andrey Markov, Markov Decision Processes (MDPs) provide a mathematical framework for modeling decision-making in situations where outcomes are partially random and partially under the control of the decision maker.

A Markov Decision Process is a discrete time stochastic control process. The process is characterized by a set of states S. At each time step the system is in a particular state, say $x \in S$. For that state there is a set of admissible actions that the decision maker may choose from , denoted by the set A_x. The outcome of any action and thus the next state of the system, is not fully predictable but can be anticipated according to the choice of action. That is, if action a is chosen to act on state $x \in S$, where $a \in A_x$, then the probability that the next state is $y \in S$ is given by $P_a(x, y)$. Finally , with each action there is an associated immediate cost that is incurred, depending both on the action and the current state of the system. For state $x \in S$ and action $a \in A_x$ the cost is given by $g_a(x)$. For the problems concerned with this project, the cost is additive over time, the problems are treated as non terminating and the aim is to choose those actions at each time step, that will minimize the total discounted infinite-horizon cost:

$$\sum_{t=0}^{\infty} \{\alpha^t g_{a_t}(x_t)\} \tag{2.1.1}$$

$$\begin{aligned} &\textit{where} & \\ x_t &\in S & \text{the state at } t \\ a_t &\in A_{x_t} & \text{the chosen action at } t \\ \alpha &\in (0,1) & \text{the discount factor} \end{aligned}$$

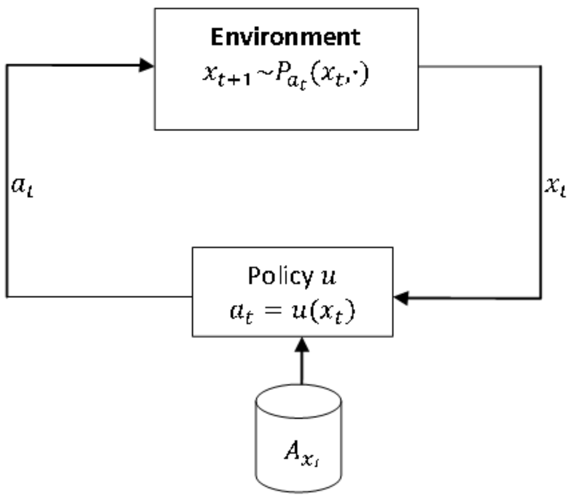

Figure 2.1: *For the present state x_t there is a set of admissible actions A_{x_t}. Policy u will select one of those actions, denoted a_t, that will shape the environment so that the next state x_{t+1} occurs according to the probability density function $P_a(x_t,\cdot)$*

The role of the discount factor in this context is to prevent the total cost from approaching infinity. However, its purpose goes beyond the mathematical need for convergence to a finite value. It also models the economical behavior that any measure of value depreciates as time progresses.

The solution of a Markov Decision Process is represented as a policy that will minimize the aforementioned cost. A policy is a mapping to an admissible (for that state) action. That is, a policy u will map state $x \in S$ to an action $a \in A_x$ so that $u(x) = a$.

The challenge in solving a MDP comes from three properties:

(a) a decision-maker has partial control of what the resulting next state will be, when the chosen action is done on the present state. That is, the decision maker cannot fully predict what the next state will be, but can control with his/her choice of action what the likelihood or the probability of a state occurring next will be;
(b) the set of possible actions depends on the present state;
(c) a cost is not only associated with an action but also with the state in which that action was performed.

As a result, an action performed now directly influences the environment within which future actions take place, and hence influences the future costs that will be incurred. Therefore any policy that solves the MDP must take into account both the immediate costs of the proposed actions, but also the probabilities associated with those actions, for a favorable state to follow.

This results in a policy that will minimize the total expected cost that will be incurred when the MDP is run. Formally, the problem that needs to be solved is:

$$\min_{u(\cdot)} E\left[\sum_{t=0}^{\infty}\left\{\alpha^t g_{u(x_t)}(x_t)\right\}|x_0 = x\right] \tag{2.1.2}$$

2.2 Dynamic Programming

A Markov Decision Process can be thought of as a collection of sub-problems. At each time-step there is a decision to be made based on the current state, so each time-step poses an independent sub-problem. The optimal policy that solves the MDP problem is the one that chooses the optimal action at each time-step. In other words, an optimal solution to the MDP can be achieved by finding the optimal solutions to all the sub-problems of the MDP[8]. This method of solving complex MDP problems by breaking them down into simpler subproblems is called Dynamic Programming.

Central to the Dynamic Programming methodology is the assignment of a value to each action for each particular state and time. That value is the sum of

(a) the immediate cost of that action, and
(b) the expected future costs that will result if that action is chosen. This cost is discounted by the discount factor α, as used in the Markov Decision Process. This cost measures how favorable a state is.

The problem of finding an optimal policy is hence reduced to the problem of choosing the particular action at each time-step, that minimizes this value. Such a policy that selects an action by comparing all actions according to a value, and then choosing the one that minimizes(or maximizes) that value, is called a greedy policy.

Assume there is a cost-to-go function $J^*(y, t_0)$, that provides the total expected cost of the MDP, if the process starts from time t_0 and from state y. The expected future cost for action a happening on state x at t_0, where $a \in A_x$, is given by :

$$\sum_{y \in S} P_a(x, y) J^*(y, t_0 + 1) \tag{2.2.3}$$

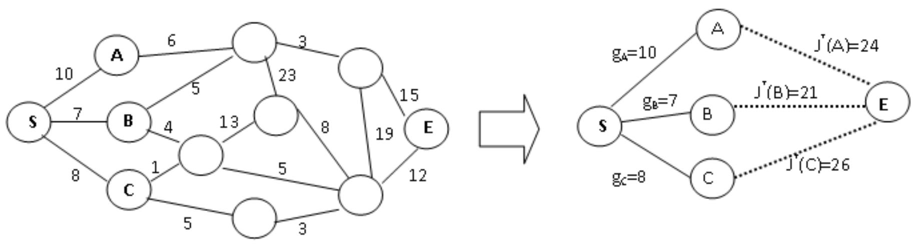

Figure 2.2: *An easy way to visualize Dynamic Programming (DP), is to consider the Minimum Cost Path problem. The aim is to travel from node S to node E with the smallest possible cost. Instead of finding a solution to the entire problem, i.e laying out the entire path from S to E before we start, DP will solve the problem node by node i.e.it will decide to go from S to A, and see how to proceed once we have got there. Thus, at the first node we choose to go to the next node that minimizes the sum of the action's immediate cost g_a and the resulting state's expected cost-to-go $J^*(y)$. To do this, we must calculate J^* for each node using Bellman's equation, so that J^* is equal to the minimum cost that will be incurred, if we were to reach node E from that particular node. That is, we must find for each node, the minimum cost path from that node to the destination. For this case this can be calculated using backtracking, that is, starting from the last node and moving backwards.*

Thus, the optimal policy $u^*(\cdot,\cdot)$ will map the particular state at that particular time to the action that solves the subproblem:

$$u^*(x,t_0) = \underset{a\in A_x}{\arg\min} \left\{ g_a(x) + \alpha \sum_{y\in S} P_a(x,y) J^*(y, t_0+1) \right\} \tag{2.2.4}$$

If the optimal solution for any subproblem is to be part of the overall optimal solution of the problem, then the cost-to-go function J^* used in (2.2.3) must calculate the cost when the overall optimal policy is used. In other words, $J^*(y,t_0)$ must reflect the minimum expected future cost that will be incurred from that state onwards, by assuming that the best possible action is chosen at each state in order to minimize that cost. Thus, J^* is defined as:

$$J^*(y,t_0) = \min_{u(\cdot,\cdot)} E\left[\sum_{t=t_0}^{\infty} \left\{ a^{t-t_0} g_{u(x_t,t)}(x_t) \right\} | x_0 = y \right] \tag{2.2.5}$$

We may also conjecture that $J^*(x,t)$ satisfies a recursive relation of the form

$$J^*(x,t) = \min_{a\in A_x} \left\{ g_a(x) + \alpha \sum_{y\in S} P_a(x,y) J^*(y,t_1) \right\} \tag{2.2.6}$$

which is no more than saying that the expected cost-to-go at time t, as given by (2.2.5), is the expected cost-to-go at time $t+1$, given that the optimal action is chosen presently, plus the immediate cost of that optimal action.

We observe that $J^*(x,t) = J^*(x,t') = J^*(x)$, for all t, t'. This means that cost-to-go equation (2.2.6) can be re-written as :

$$J^*(x) = \min_{a \in A_x} \left\{ g_a(x) + \alpha \sum_{y \in S} P_a(x,y) J^*(y) \right\} \tag{2.2.7}$$

This equation is called Bellman's equation [9]. By noting that the transition probabilities $P_a(x,y)$ do not depend on time and taking into account the time-independent definition of the cost-to-go function in (2.2.7), it is inferred that the optimal policy as defined in (2.2.4) does not depend on the current time stage t, and $u^*(x,t) = u^*(x)$ for some policy $u^*(\cdot)$. Such a policy that does not depend on time stage is called stationary policy[1]. The equation of the optimal policy (2.2.4) becomes:

$$u^*(x) = \arg\min_{a \in A_x} \left\{ g_a(x) + \alpha \sum_{y \in S} P_a(x,y) J^*(y) \right\} \tag{2.2.8}$$

To conclude, Dynamic Programming solves a Markov Decision Process by considering the subproblem of each time-step separately. The problem of finding an optimal policy for the entire MDP reduces to the problem of assigning a cost-to-go value J^* to each state. The cost-to-go value reflects the expected cost-to-go if the optimal policy for the MDP is followed. The cost-to-go is given as in (2.2.5), and it is the unique solution of Bellman's equation [3, Vol.2,Chap.1] [4, Chap. 2] [9]. Once we solved Bellman's equation to find J^* for all states, an optimal policy may be defined as the greedy policy. A greedy policy maps a state to an action, by going through all admissible actions and chooses the one that minimizes the sum of the immediate cost $g_a(x)$ with the (discounted) expected future cost $\alpha \sum_{y \in S} P_a(x,y) J^*(y)$ for that action, or formally, by solving (2.2.8).

2.3 Approximate Dynamic Programming

During the development of the Dynamic Programming methodology Bellman realized that it is not always possible to obtain an exact numerical solution to Bellman's equation . In fact, he concluded that for the majority of real life

[1]For more rigorous analysis and proofs, see [8], [3, Vol.2,Chap.1] and [4, Chap. 2]

problems, it was computationally infeasible to find the true cost-to-go cost for each state. Bellman dubbed this limitation as 'the curse of dimensionality'[8].

The 'curse of dimensionality states' that in real life practical problems the size of a state space grow exponentially with the number of state variables. This limitation prevents the use of exact dynamic programming in various real life applications. Consider the example where there are n dimensions in a state (i.e. n different properties that make a state unique from the rest). If the state space is discretized so that there are d discrete points in each state dimension, then there are d^n possible states. In order to solve Bellman's equation numerically, we need to manually calculate the right hand side of the equation for all d^n states. The calculations become even more complex according to the number of available actions for each state A_x.

To overcome this limitation an entirely new branch of Dynamic Programming has been developed, called Approximate Dynamic Programming. This area of DP aims to find sub-optimal solutions by means of simulation based optimizations, Monte Carlo sampling techniques and open-loop-feedback controls [Vol.2,Chap.6][3][Vol.1,Chap.6][3][10][4] . Instead of looking for an exact solution J^* to Bellman's, we look for an approximation $\widetilde{J}$ that is as close to J^* as possible.

Approximate Dynamic Programming includes numerous methods and their variations, because none of the methods is considered to be applicable in every situation. The choice of a method is highly problem dependent and success is not guaranteed [10]. For this project we are interested in two widely used techniques of Approximate Dynamic Programming, described below.

2.3.1 State Augmentation and Parametric Cost-To-Go approximation

In classical (exact) DP the aim is to find the cost-to-go value for each state. Under the curse of dimensionality, such a calculation explodes exponentially in practical problems with complex state systems. One technique for overcoming the curse of dimensionality, is to generate an approximation of the cost-to-go function with a parametrized class of functions. To produce such an approximation :

1. A class of K basis functions $\phi_1, \ldots, \phi_K$ must be defined, with each basis function ϕ mapping a particular characteristic of the state variables into a real number $\phi : S \mapsto \Re$. The basis functions essentially augment a state and reduce it to a set of K characteristics that are considered relevant to the optimization;

2. A weighting vector $r \in \Re^K$ must be computed for the class, so that by weighting each basis function according to r, the weighted sum of the parametrized class of functions $\widetilde{J}(\cdot, r)$ approximates the optimal cost-to-go function J^* as closely as possible:

$$\widetilde{J, r} = \sum_{k=1}^{K} r_k \Phi_k \approx J^*$$

Choosing suitable basis functions is a very problem-specific task and requires practical experience or theoretical analysis that provides rough information on the shape of the function to be approximated. Even though some mathematical guidelines exist, the basis functions are usually hand crafted based on whatever human intelligence,insight,or experience is available [4, chap. 6][10].

Given a choice of basis functions $\phi_1, \ldots, \phi_K$ a matrix may be defined:

$$\Phi = \begin{bmatrix} | & & | \\ \phi_1 & \vdots & \phi_K \\ | & & | \end{bmatrix}$$

The aim is now to find a weight vector $\widetilde{r} \in \Re^K$ such that $\Phi\widetilde{r}$ is a close approximation to the optimal cost-to-go J^*.

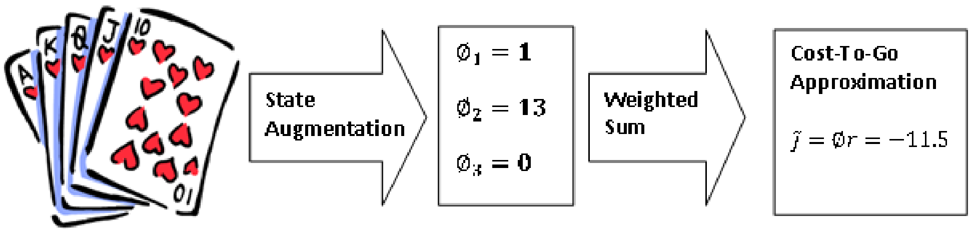

Figure 2.3: *Consider a policy that plays a game of poker. The initial state is the suit and the value of each of the five cards that we currently hold. The state is then augmented using three basis functions :* Φ_1 *the position of the card combination in hierarchy = 1,* Φ_2 *the value of the highest card = 13 in our hand, and* Φ_3 *the number of similar or higher card combinations possible =0. The cost-to-go is then approximated, by summing up the three values according to a weight vector* $r^T = (1.5, -1, 3)$*, thus, the approximated cost-to-go* $\widetilde{J} = -11.5$

2.3.2 Simulation Based Optimization

Another technique widely used in Approximate Dynamic Programming is the employment of simulators to collect samples that we base our cost-to-go estimation on. Such a technique is used when it is not possible to describe the system analytically, but it is easy to simulate the system [11],[Vol.1,Chap.6][3]. This situation arises when we are dealing with an intractable number of states or when it is infeasible to list all possible outcomes for each particular state and for every particular action. It is also used in cases where information about the model, such as transitional probabilities, is not fully available until up to the point when the control must be applied. Simulation Based Optimization is often used in conjunction with parametric cost-to-go approximations and it is usually an iterative procedure where the following steps are followed:

(a) Set an initial estimation of the cost-to-go function. The cost-to-go function can be (but not necessarily) approximated using parametric approximation as describe above.
(b) Using the given cost-to-go estimation in the greedy policy, simulate a run of the system and record the necessary parameters of the system in order to generate samples. The samples are usually generated according to a distribution.
(c) Using the samples generated above, the optimization process is applied in order to find a new estimation of the cost-to-go function.
(d) The simulation may be repeated and the parameters refined at each iteration, until reasonable performance is achieved.

2.4 Solving Bellman's Equation

Dynamic Programming reduces the problem of finding an optimal policy, to the one of solving Bellman's equation for all possible states of the system:

$$J^*(x) = \min_{a \in A_x} \left\{ g_a(x) + \alpha \sum_{y \in S} P_a(x,y) J^*(y) \right\} \tag{2.4.9}$$

There are several methods of solving this equation. Most methods are designed to work with approximate dynamic programming, as it has a wider application in real life. We consider two methods described below.

2.4.1 Linear Programming Approach

Linear programming is a method for minimizing or maximizing a linear objective function, subject to a set of linear equality or inequality constraints. A linear

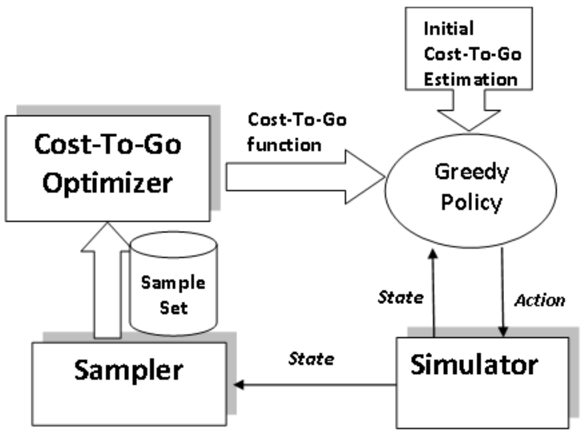

Figure 2.4: *An iterative simulation based optimization. There are three major components. The simulator, that operates according to the current greedy policy, the sampler that decides when to record the state, and the cost-to-go optimizer, that uses the sample set to generate a cost-to-go that is optimized for the given sample. The cost-to-go estimation, is then used to generate the greedy policy that the simulator uses. In this case, the simulation is repeated until reasonable performance is achieved. However, this is not necessary, and optimization may involve a single step simulation*

problem is usually of the form:

$$\begin{aligned} min \quad & c^T x \\ s.t. \quad & Ax \leq b \end{aligned}$$

The objective function is given by $c^T x$ where x is a vector of the decision variables which we look to find, c is the cost vector, that is the known coefficients that each decision variable has in the objective function. A is the matrix of known coefficients, and the system of inequalities expressed as $Ax \leq b$ is the set of linear constraints that the optimal solution of the objective function must satisfy. Any minimization problem can be transformed to a maximization problem, by using the equality

$$min \quad c^T x = -max \quad -c^T x$$

There are several techniques for solving a linear problem, and current solvers can handle problems that involve millions of constraints and thousands of decision variables.

Linear Programming is used in Dynamic Programming to solve Bellman's equation 2.2.7. Bellman's equation can be reformulated as a Linear Problem

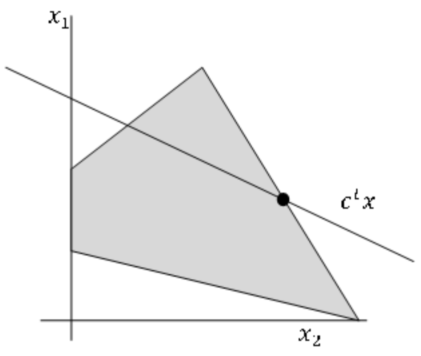

Figure 2.5: *The graph shows a two dimensional linear minimization problem. The decision variables are x_1 and x_2 and form the axis of the graph. The shaded area shows the feasible region, that is, all possible values of x_1 and x_2 that satisfy $Ax = b$. The black dot is the point where the optimal value resides. It is the lowest possible value that the objective function $c^T x$ can take, and remain feasible (i.e. within the shaded area)*

(LP) as:

$$\begin{aligned} max_J \quad & c'J \\ s.t. \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x,y)J(y) \geq J(x) \\ & \forall x \in S, a \in A_x \end{aligned} \tag{2.4.10}$$

where c' is a vector with positive components, signifying the state-relevance weights. By the constraints definition , $J(x)$ must be smaller than the left hand side, for all actions possible in that state. This includes the action that is optimal, so the constraints require that:

$$J(x) \leq min_{a \in A_x} \left\{ g_a(x) + \sum_{y \in S} P_a(x,y)J^*(y) \right\} \forall x \in S \tag{2.4.11}$$

But the right hand side of this inequality is identical to the true cost-to-go value (2.2.6) J^*. Thus any feasible J satisfies $J \leq J^*$, which in turn implies that the objective function of the LP can only increase up to the point where it is equal to J^* for any positive weight vector c, and so the solution of the LP is equal to J^* the unique solution to Bellman's equation (2.2.7).

2.4.2 Least Square Approach

Least Square methods are methods that are usually used in statistical contexts as a mean of regression analysis. Least square methods attempt to derive a function that fits the data as closely as possible.

In the context of dynamic programming least squares are used along with iterative simulation techniques, in order to derive an approximation for the cost-to-go function. That is, using simulation to run the system under a policy, a large sample of states and actions is generated. Using that sample set, we attempt to fit a cost-to-go approximation so that it solves Bellman's equation for the specific sample.

Recall that Bellman's equation (2.2.7) is defined as:

$$J^*x = \min_{a \in A_x} \left\{ g_a(x) + \alpha \sum_{y \in S} P_a(x,y) J^*(y) \right\} \quad \forall x \in S. \tag{2.4.12}$$

The Least Square method adjusts J so that the left hand side of the equation is as close as possible as the right hand side, for the specific sample of states. Namely,given a sample $\bar{X}$ the Least Square problem that we need to solve is:

$$\min_J \| J(x) - \left(\min_{a \in A_x} \left\{ g_a(x) + \alpha \sum_{y \in S} P_a(x,y) J(y) \right\} \right) \|^2 \tag{2.4.13}$$

for some choice of a norm. The methods that rely on a Least Square are usually one step iterations, meaning that they approach a near optimal solution in one step. There are several variations however, that control the conversion rate to avoid being trapped in local minima. Some examples of Least Square methods are:

(a) Least Square Policy Evaluation (LSPE)
(b) Policy Evaluation by Projected Vale Iteration (PVI)
(c) Least Square Temporal Differences (LSTD(λ))

CHAPTER 3

The BRLP method

The Bootstrapped Reduced Linear Programming (BRLP) approach to Approximate Dynamic Programming is a state of the art method for solving a parametric approximation of Bellman's equation. The method is designed to overcome the curse of dimensionality. It relies on simulation based optimization of the approximate cost-to-go function and iterative refinement of the solution.

The method works as follows:

1. Guess an initial weight vector $r_0 \in \Re^K$ to approximate the cost-to-go function using the class of K basis functions Φ
2. Use the cost-to-go approximation Φr_k to generate a greedy policy u_k
3. Perform a simulation using the current policy
4. Throughout the simulation, record the state of the system at specific time intervals to generate a sample $\overline{X_k}$ that is associated with current policy u_k. The sampling intervals must be far apart, so that the samples generated are independent and identically distributed.
5. Solve a specific linear problem (RLP) conditioned on the sample $\overline{X_k}$ to generate a new weight vector $r_{k+1} \in \Re^K$
6. Increment k and go to step 2

The success of the method relies on two observations:

(a) The linear programming approach can be used to generate a descend parametric approximation to the cost-to-go function;

(b) The sampling technique generates samples that are representative of the entire system, and the linear programming optimization can be based on them instead of the entire system, without a significant loss of precision in the solution.

3.1 Linear Programming Approach

In exact Dynamic Programming, Linear Programming can be used to find the cost-to-go function J^* for all states by solving the LP problem (see section 2.4.1):

$$\begin{aligned} \max_{r} \quad & c'J \\ \text{s.t.} \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x,y) J(y) \geq J(x) \\ & \forall x \in S, a \in A_x \end{aligned} \tag{3.1.1}$$

However, the problem's formulation requires one decision variable per state and one constraint per state-action pair. For real life systems that involve a large amount of states and thus suffer from the curse of dimensionality, a problem formulated as above becomes prohibitively large to be solved exactly. The BRLP method relies on a parametric approximation for the cost-to-go function. The cost-to-go is approximated parametrically, using a class of K basis functions Φ (see section 2.3). The LP problem is reformulated to find the weight vector $r \in \Re^K$ so that $\Phi r \approx J^*$. The LP problem (3.1.1) is modified by substituting the exact cost-to-go function J with its linear approximation Φr . The problem that needs to be solved now becomes

$$\begin{aligned} \max_{r} \quad & c'\Phi r \\ \text{s.t.} \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x,y) (\Phi r)(y) \geq (\Phi r)(x)) \\ & \forall x \in S, a \in A_x \end{aligned} \tag{3.1.2}$$

This is called the approximate linear program (ALP). Its solution is a weight vector $\widetilde{r} \in \Re^K$ that will result in a good cost-to-go approximation $\widetilde{J}(\cdot, r)$. Observe that the optimization variables no longer depend on the state space, but are equal to the number of basis functions K regardless on how large the state space is. This is a dramatic step towards tractability, however, the number of constraints is still proportional to the number of state variables and the problem remains infeasible.

3.1.1 State Relevance Weights in ALP

Recall that in exact DP (section 2.2), the state-relevance vector c in the objective function does not affect the optimal solution, and the solution J^* is the optimal solution of the exact LP for any cost vector $c > 0$. This is not the case for the ALP.

Let

$$\|J\|_{1,c} = \sum_{x} c(x)J(x)$$

denote a weighted l_1-norm. If J^* is the exact solution to Bellman's equation, a solution of the ALP (3.1.2) is shown to minimize the weighted approximation error

$$\|J^* - \Phi r\|_{1,c}$$

within the feasible region of the ALP and with state relevance weights equal to c. That is, a solution to the ALP is also a solution to the problem:

$$\begin{aligned} \min_{r} \quad & \|J^* - \Phi r\|_{1,c} \\ s.t. \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x,y)(\Phi r)(y) \geq (\Phi r)(x)) \\ & \forall \in S, a \in A_x \end{aligned}$$

This suggests that c imposes a trade off in the quality of the approximation across different states. By assigning higher values in c for a particular region in the state space, the algorithm can be directed in generating a better approximation for that region.

Let J_u be the expected discounted infinite-horizon cost incurred when using a greedy policy generated by cost-to-go function J. Let J^* be the expected cost when the optimal policy is used, i.e. the greedy policy generated by the true cost-to-go function J^*. Then $\|J_u - J^*\|_{1,v}$ is the increase in the expected cost as a result of using the suboptimal greedy policy u, and conditioned on the initial state of the system being distributed according to a probability distribution v. It can be shown that this loss in policy performance is bounded according to:

$$\|J_u - J^*\|_{1,v} \ \leq \ \frac{1}{1-a}\|J - J^*\|_{1,\mu_{u_j,v}} \qquad (3.1.3)$$

where $\mu_{u_j,v}$ is the probability distribution that captures the expected frequency of visits to each state when the system runs under greedy policy u_j[12]. As (3.1.3) shows, the loss of performance due to the use of sub-optimal greedy policy generated by the approximate cost-to-go function J, is bounded by the weighted approximation error of J to J^*, where the weights are equal to the frequency $\mu_{u_j,v}$. In other words, if the ALP can generate a cost-to-go function J that is close to J^* for those states that are expected to be visited more when a system runs under a near-optimal policy, then the greedy policy generated by J is close in performance to the optimal policy. Combining this with the previous observation that c can direct the algorithm to generate a good approximation for specific

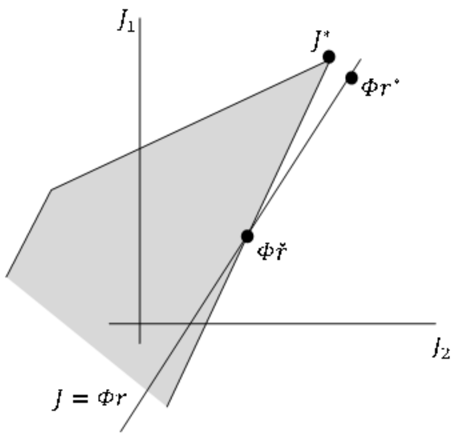

Figure 3.1: *Graphical interpretation of approximate linear programming*

states, it is obvious that c should mimic the frequency with which different states are expected to be visited, when the system runs under a near-optimal policy.[1]

3.1.2 Error Bounds

The ALP solution is a weight vector $\widetilde{r} \in \Re^K$, that will result in a good cost-to-go approximation $\Phi\widetilde{r}$. If the optimal cost-to-go function lies within the span of the basis functions,that is the basis functions are chosen as to be capable to represent the exact cost-to-go function J^*, then the ALP (3.1.2) will yield the exact optimal cost-to-go function,i.e. $\Phi\widetilde{r} = J^*$. However, it is difficult in practice to select the basis functions that contain the optimal cost-to-go solution. Consider the illustration in Figure 3.1. The figure illustrates a MDP process with two states 1 and 2, and the plane presented is the space of all cost-to-go functions over the state space. The shaded area shows the feasible region of both the exact (3.1.1) and the approximate (3.1.2) LPs. J^* is the solution of the exact LP, thus the true cost-to-go function. The subspace denoted by the line $J = \Phi r$, is the span of the basis functions. This span comes very close to the true cost-to-go function, with the best cost-to-go approximation being at $\Phi^* r$, given this choice of basis functions. However,the solution of the approximate LP is restricted by the constraints to be within the feasible space,so the solution of the approximate LP results in the approximate cost-to-go approximation $\Phi\widetilde{r}$.

The distance between the obtainable cost-to-go approximation $\Phi\widetilde{r}$ and the exact cost-to-go J^*, is not much greater than the distance between the best possible approximation $\Phi^* r$ and the exact cost-to-go J^*. A naive bound on the

[1]For more details and proof see [12, p.5]

approximation error of ALP's solution relative to the minimum possible error (given the choice of basis function), is given by:

$$\|J^* - \Phi\widetilde{r}\|_{1,c} \;\leq\; \frac{2}{1-\alpha} \min_r \|J^* - \Phi r^*\|_\infty$$

This is a very loose bound since $\|J^* - \Phi r^*\|_\infty$ is likely to be very large for large scale spaces. However, better bounds exist with more sensible measures, that ensure that given a good choice of basis functions and state-relevance weights,the ALP will give a good approximation[2].

3.2 Constraint Sampling and ALP

The approximate linear programming method is a significant improvement on the number of optimization variables. The ALP problem (3.1.2) requires only K objective variables, where K is the number of basis functions Φ, eliminating the need for one variable per state, as is the case in the exact LP (3.1.1). However, the constraints of the ALP are still proportional to the state space dimensions (one constraint per state-action pair) and thus are susceptible to the curse of dimensionality, rendering the ALP intractable for most practical problems. The BRLP method utilizes one method for creating a tractable approximation of the ALP, the reduced linear program (RLP).

Reduced Linear Programming is based on the general assumption that when the number of constraints exceeds by some orders of magnitude the number of optimization variables, then most constraints are either inactive or have a minor impact on the feasible region. Therefore, it can be speculated that the feasible region specified by all constraints can be closely approximated by a subset of these constraints.

Using a sampling distribution ψ, we sample from the original problem a set of N constraints, denoted by $\overline{X}$. We then solve the ALP for that set, thus the problem (3.1.2) becomes the RLP:

$$\begin{aligned} \max_r \quad & c'\Phi r \\ \text{s.t.} \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x,y)(\Phi r)(y) \geq (\Phi r)(x) \\ & \forall (x,a) \in \overline{X} \end{aligned} \tag{3.2.4}$$

[2]For more details see [12, p.7] and [6]

The use of constraint sampling is motivated by the findings of [13].Let the number of optimization variables of the problem be K, that is the number of basis function. Let the size of our constraint sample set $\overline{X}$ be:

$$\begin{aligned} N &\geq \frac{4}{\epsilon}(K \ln \frac{12}{\epsilon} + \ln \frac{2}{\delta}) \\ \text{for} & \quad \delta, \epsilon \in (0,1) \end{aligned} \tag{3.2.5}$$

If the reduced LP (RLP) (3.2.4) is solved instead of the original ALP (3.1.2), the constraints of the original problem that will be violated by the solution of the RLP are unimportant [13]. In order to measure the importance of a set of constraints, we use the sampling distribution ψ. The distribution ψ now assumes a dual role:

1. The distribution with which we sample the constraints.
2. A measure of the quality of a particular set of constraint.

If V is the set of the constraints that are active in the original problem but will not be selected during the sampling procedure, that is, V is the set of the constraints that will be violated by the solution of the RLP, then $\psi(V) \leq \epsilon$ with a probability of at least $1 - d$. In other words, the importance (as measured by ψ) of the violated constraints V will be less than ϵ, with probability at least $1-\delta$. This is an important result because it states that the number of constraints that we need to sample does not depend on the number of constraints that the original problem has. Instead, this number depends on the quality of the solution we wish to find. That is, from which point onwards (equal to the ϵ parameter) a constraint is considered important and needs to be included in the RLP and with what probability (equal to $1 - \delta$) we want to guarantee their inclusion.

In spite of the above result it is still not guaranteed that a solution to an RLP will be close to that of the ALP. In particular it might be the case that only a few constraints in the ALP affect the solution space, as is the case in figure 3.2. Removing constraints at random in such a problem might dramatically change the solution of the problem. However, the structure of the ALP prevents this from happening and it is established that , when the state-action pairs are sampled independently according to the distribution:

$$\psi^*_\alpha(x) = (1-\alpha)E\left[\sum_{t=0}^{\infty} \alpha^t 1 x_t = x | x_0 \sim c, a_t = u^* x_t\right]$$

then for a large enough sample set there is a high probability that the approximation error of the RLP's solution will be match closely the approximation

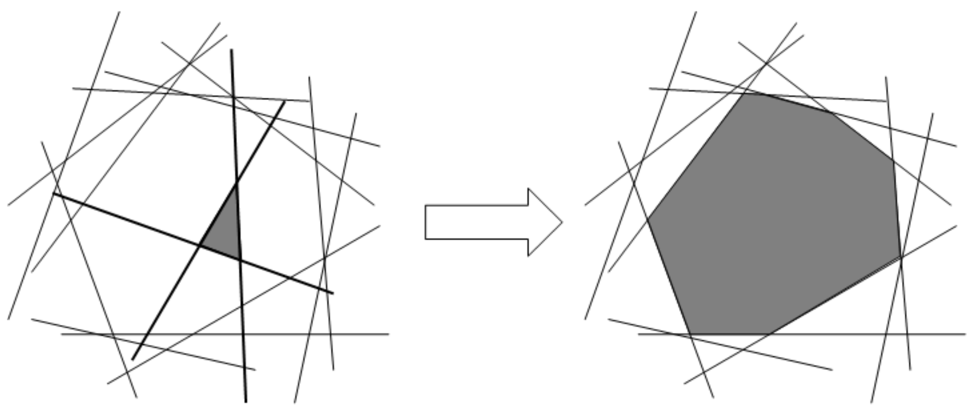

Figure 3.2: *The shaded area is the feasible region. This formulation exhibits a large number of redundant constraints, which can be removed. However, if we remove either of the three active constraints will cause the feasible region to change significantly. Fortunately, the structure of the ALP problem ensures that not only there are a lot of redundant constraints that can be removed, but also that even if we remove some key constraints the feasible region will not change much*

error of the ALP's solution[13]. That is, for large N, if $\Phi\widehat{r}$ is the approximate cost-to-go generated by the RLP, $\Phi\widetilde{r}$ is the approximate cost-to-go generated by the ALP, and J^* is the true cost-to-go, then there is a good probability that $||J^* - \Phi\widehat{r}||_{1,c} \approx ||J^* - \Phi\widetilde{r}||_{1,c}$. In other words a solution of the RLP will yield a cost-to-go approximation almost as good as the one resulting from the solution of the ALP.

The disadvantage of this result, is that the sampling distribution ψ^*_α requires the knowledge of the optimal policy u^*. However, empirical evidence [6][7] show that this result holds for distributions that are close to ψ^*_α, and thus RLP provides a tractable means for providing a meaningful approximation to J^*.

3.3 The Bootstrapped RLP

In the previous sections, was shown that one can create a good approximation of the optimal cost-to-go function J^*, by making suitable choice of basis functions Φ and by solving the tractable RLP defined in (3.2.4):

$$\begin{aligned}\max_r \quad & c'\Phi r \\ \text{s.t.} \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x,y)(\Phi r)(y) \geq (\Phi r)(x) \\ & \forall (x,a) \in \overline{X}\end{aligned}$$

. For this solution to be meaningful though, great care needs to be taken for:

1. defining the state-relevance weight c so that it mimics the frequency with which each state will be visited when the system runs under the optimal policy
2. defining the sample distribution ψ so that is captures the distribution with which state-action pairs occur when the system runs under optimal policy

The correct definition of these two parameters is crucial because, as demonstrated (subsection 3.1.2, section 3.2), it affects both the approximation error of the derived approximate cost-to-go function, but also the performance of the greedy policy generated using that approximate cost-to-go function.

Now, assume that we run the process under a near optimal policy. For a choice of a sampling interval M, we record the state visited by that policy at time stages multiple of M and we incorporate it in the state sample $\overline{X}$. By doing so, we are generating nearly i.i.d samples of states, distributed according to the relative frequencies with which states are visited by the near optimal policy. We then use this sample to solve the following RLP:

$$\begin{aligned}\max_r \quad & \sum_{x \in \overline{X}} \Phi r \\ s.t. \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x,y)(\Phi r)(y) \geq (\Phi r)(x) \\ & \forall (x) \in \overline{X}, \forall a \in A_x\end{aligned} \tag{3.3.6}$$

Observe that in the above RLP:

1. c is replaced by the sampling distribution. Since the sampling distribution is equal to the frequency with which states are visited by the near-optimal policy, c now mimics that frequency, which is what we want;

2. for each sample state, a constraint is generated for every possible action in that state. It is because the policy used to sample the constraints is sub-optimal, thus might not choose the optimal actions at each step. By including all actions, the optimal action is guaranteed to be part of the constraints.

The solution of the RLP (3.3.6) might generate a superior policy than the one it started with(although no guarantees exist). In this light, we can use the new superior policy to create another sample of states, and then proceed to solve (3.3.6) to obtain an even better policy. As a result, a bootstrapping algorithm is defined as follows:

1. Begin with a simulator that uses a policy u_0
2. Generate a sample $\overline{X_k}$ using policy u_k
3. Solve the RLP based on the sample $\overline{X_k}$ to generate a policy u_{k+1}
4. Increment k and go to step 2

Things to note on this algorithm are that :

(a) It depends on an initial policy u_0. An initial policy may be created by guessing and adjusting the weights r, until reasonable performance is achieved.
(b) There is no guarantee that the policy generated at each iteration will be better than the one generated in the previous iteration. This is because all policies are near-optimal, whilst for the approximation errors of the RLP to be bounded, we require that a sampling distribution is generated using the optimal policy. Evidently, an iteration may be repeated several times, before a better policy is generated.

3.4 Extensions to the BRLP method

One expects that successful iterations of the BRLP method will result to continually improving policies. However, experimental results [6][7] show that the performance of policies generated by the BRLP peaks, usually after three to four iterations of the BRLP, and then dies, as shown in figure 3.3. In the original paper [6] one extension is proposed and tested without success. In order to examine this further, we have devised some more extensions and tested to see how they perform.

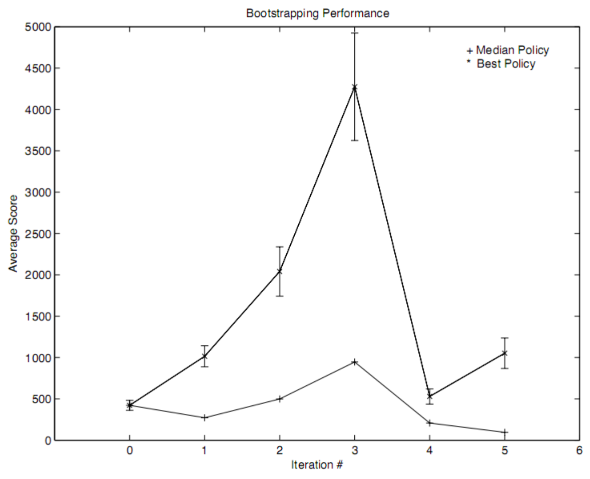

Figure 3.3: *The performance of the BRLP method peaks, as demonstrated in [6].*

3.4.1 Guarded Sampling

This is the extension proposed in the original BRLP paper [6]. The BRLP method is modified, so that the state sampling distribution in a given iteration is the average of the distribution induced by the latest policy and the distribution used in the previous iteration. That is, we construct our sample set $\overline{X_k}$ using samples generated by the current policy u_k and samples that belong to the previous set $\overline{X_{k-1}}$.

3.4.2 Adaptive Sampling

This extension is to investigate whether the drop in performance is due to the sampled states becoming more dependent on each other, as the policy improves in performance. This extension will test if recording the states at a fixed time interval, generates states that become more and more dependent as the policy gets better. The intuition behind this assumption, is that as the policy becomes better, it can 'look' further into the future, and thus sampled states need to be further apart in order to be independent and identically-distributed as required by the BRLP. Thus, to combat such a situation, we increase our sampling interval according to the percentage increase in policy performance. To do this, we need to first gage the performance of a policy, by running it for some time before we can start sampling.

3.4.3 Conditional Sampling

This extension is inspired by Semi-infinite Linear Programming. Semi-infinite LP deals with LPs that have an infinite amount of constraints. Instead of sampling the constraints, as we do with the BRLP, Semi-infinite programming starts with a small number of constraints and finds a solution. Using that solution, it then attempts to find the worst offender, that is, the constraint that is violated the most by the proposed solution. The constraint is then added to the LP, and the procedure is repeated [14]. To use such a technique in the BRLP method, we are going to run the BRLP until we see a drop in performance. Let u_k be the best policy that was generated by the BRLP. That policy is generated by sample X_{k-1}. What we propose is, to use the u_k policy in order to find an offending constraint. That is, we will run the simulator , but we will only sample the states where the constraints are violated. That is, if we are at state x, and policy u_k selects an action a so that it violates the condition :

$$g_a(x) + \alpha \sum_{y \in S} P_a(x, y)(\Phi r)(y) \geq (\Phi r)(x) \tag{3.4.7}$$

then we record that state. The recorded states are then incorporated to the previous sample X_{k-1}, and the RLP is resolved based on that new sample.

CHAPTER 4

Tetris Optimization

4.1 Problem Specification

4.1.1 Description

Tetris is a video game in which falling bricks are positioned on a two dimensional grid of a given width and height.The pieces may be rotated and positioned at any point on the wall.The set of possible shapes for the falling pieces is finite. For a typical game, there are 7 different pieces composed of four square blocks each, called tetrominoes. The game starts with an empty grid. A point is received for each row constructed without any holes, and the corresponding row is cleared. The game ends once the height of the wall crosses a particular threshold. The objective is to maximize the expected number of points accumulated over the course of the game.

Tetris is a typical one-player game that can be formulated as a MDP. Because it suffers from the curse of dimensionality, Tetris became very popular in testing several Approximate Dynamic programming methodologies, including the BRLP method. As a result, Tetris can be used as a benchmark for comparing the different methodologies and establishing their performance.

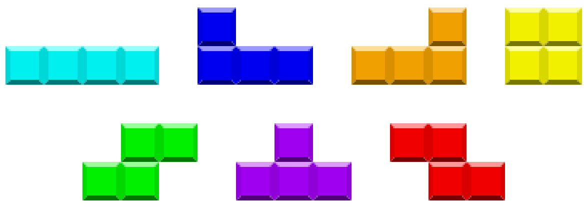

Figure 4.1: *The seven possible pieces of Tetris.*

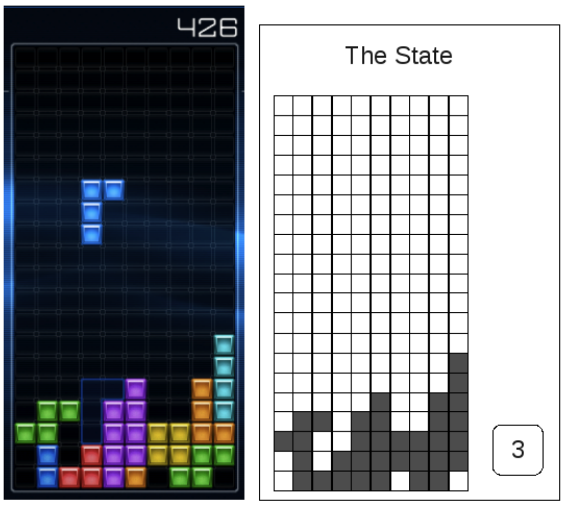

Figure 4.2: *A typical Tetris game along with the MDP representation of its state. The state has two components, the binary board configuration and a number identifying the falling piece. In this case number 3 represents the current falling piece 'J'*

4.1.2 Tetris as a MDP

Tetris can be formulated as a Markov Decision process:

1. A Tetris state is represented using two components:
   (a) The board configuration, that is, a binary description of the full/empty status of each square in the grid (before the current piece is placed)
   (b) The shape of the current falling piece.

   For a standard Tetris game of a board with dimensions 20(rows) $\times$ 10(columns) and 7 possible pieces, there are

   $$2^{20\times10} * 7 \approx 1.125 \times 10^{61}$$

   possible states. It is evident that the state space is large enough to prevent the Tetris MDP from being solved using exact DP.
   Tetris is a non-deterministic MDP.The decision maker has partial control on the next state. The player can control through his/her choice of action the first component of the state,i.e. what the board will look like in the next run. However, the player cannot control the second component of the next state, that is, the shape of the falling piece, which is chosen randomly from a finite set of possible shapes.

Piece	Rotations	Translations	Total Actions
'Z'	2	8,9	17
'S'	2	8,9	17
'L'	4	9,8,9,8	34
'J'	4	9,8,9,8	34
'T'	4	9,8,9,8	34
'I'	2	10,7	17
'O'	1	9	9
Average			23.14

Table 4.1: *The table shows the possible actions for each falling piece. The translations shows number of different possible columns that a piece may be placed on the board, according to its orientation. The number of possible orientations (rotations).*

2. For a particular state, the possible actions are all the available combinations of the horizontal positioning and rotation applied to the falling piece. We assume that the piece can be rotated and moved freely, that is, the actions available for a particular piece are not restricted by the board (i.e. when the wall on the board is too tall). The action space only depends on the falling piece, and it contains all possible combinations of rotations and horizontal translations for that piece. For the standard set of pieces, the number of available actions per piece is shown in Table 4.1.2.
3. An immediate cost can be associated with each action acting on each state. For example a reward can be defined to be equal to the number of points gained as a result of the action happening on the current state, and the MDP optimization objective is to maximize the expected sum reward over the course of the game. Another formulation might be that an action cost is the average height of the wall as a result of the action happening on that state. The objective will then be to minimize the expected sum of discounted future costs. Empirical evidence [6] suggests the second formulation results in better policies.

4.1.3 Approximate cost-to-go function for Tetris

Given a board width w and height h, and a set of p possible shapes for the falling piece, it is easy to see that there are $p2^{w \times h}$ state variables. Thus , exact Dynamic Programming cannot be applied here, and so an approximation to the cost-to-go function needs to be designed.

The cost-to-go function of the Tetris can be defined using a linear combination over a class of basis functions Φ (see sections 2.3). The following choice of basis functions has been proposed by [3] and was successfully used in several occasions [3][6][7][5]. For a board with width w and height h, $2w + 2$ basis function can be defined as:

(a) w basis functions, mapping the state to the height h_k of each of the w columns
(b) $w - 1$ basis functions,each mapping the stat to the absolute difference between the heights of successive columns:$|h_{k+1} - h_k|, k = 1, \ldots, w - 1$
(c) 1 basis function that maps the state to the maximum column height $max_k h_k$
(d) 1 basis function that maps the state to the number of 'wholes' in the wall
(e) 1 basis function that is equal to one at every state, serving as a constant offset

4.1.4 Performance Objectives

Below is a table comparing the results from policies obtained for different state of the art algorithms. All policies use the same class of basis functions. The game has the standard board size of 20×10 ($h \times w$), and standard set of possible pieces (shown in figure 4.1).

Algorithm	Mean Score	Computation Time	Reference
Policy Iteration	3183	hours	[4]
Bootstrapped RLP	4274	hours	[6]
CE+RL	21,252	?	[5]
CE+RL,decreasing noise	348,895	months	[5]

It is shown that the BRLP method can produce a reasonably good result within a reasonable time period [6]. The objective of this part of the project is to reproduce the performance achieved by the BRLP. Given a standard Tetris board of 20 rows times 10 columns and the standard set of the 7 shapes for the falling pieces (figure 4.1, we use the class of 22 basis functions Φ as defined in section 4.1.3. A weight vector $\widehat{r} \in R^{22}$ must be calculated using the BRLP method, so that the approximate cost-to-go function given by $\Phi\widehat{r}$ generates a greedy policy that can achieve an average score over a 100 games of approximately 4000 lines per game.

4.2 Problem Formulation

The RLP to be solved is (from section 3.2)

$$\begin{aligned} \max_{r} \quad & \sum_{x \in \bar{X}} \Phi\, r && (4.2.1) \\ \text{s.t.} \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x,y)(\Phi r)(y) \geq (\Phi r)(x) \\ & \forall (x,a) \in \bar{X} \end{aligned}$$

where $\Phi \in \Re^{|S| \times K}$ is the matrix constructed by the basis functions $\phi_k,\ k = 1 \ldots K$ (see section 2.3). Each basis function $\phi_k,\ k = 1 \ldots K$ maps a state $x \in S$ to a real value, that is $\phi_k(x) \in \Re,\ k = 1 \ldots K$. The weight vector $r \in \Re_K$ is used to sum up the basis functions ϕ_k to get the cost-to-go approximation $\widetilde{J}$. For any single state $x \in S$:

$$\begin{aligned} \widetilde{J}(x) &= (\Phi r)\ (x) \\ &= \sum_{k=1}^{K} r_k \phi_k(x) \\ &\equiv (\,\Phi(x)\,) r && (4.2.2) \end{aligned}$$

Using (4.2.2) we can rewrite our objective

$$\sum_{x \in \bar{X}} (\Phi r)\ x \equiv (\sum_{x \in \bar{X}} \Phi(x)\,) r \qquad (4.2.3)$$

With the same reasoning, the constraints are re-written:

$$\begin{aligned} & g_a(x) + \alpha \textstyle\sum_{y \in S} P_a(x,y)(\Phi r)(y) && \geq (\Phi r)(x) \\ \equiv\ & g_a(x) + \alpha(\textstyle\sum_{y \in S} P_a(x,y)\Phi(y)\,) r && \geq (\,\Phi(x)\,) r \\ \Leftrightarrow\ & (\Phi(x)) r - \alpha(\textstyle\sum_{y \in S} P_a(x,y)\Phi(y)) r && \leq g_a(x) \\ \Leftrightarrow\ & (\,\Phi(x) - \alpha \textstyle\sum_{y \in S} P_a(x,y)\Phi(y)\,)\ r && \leq g_a(x) \qquad (4.2.4) \end{aligned}$$

There are two components in the state:

(a) The board configuration
(b) The shape of the falling piece

The board configuration evolves in a deterministic manner. The decision maker has absolute control how the board will look at the next time step by choosing where to place the current falling piece. However, the player has no control what the next falling piece will be. Any state y following state x and action a, must have as a board configuration the board that resulted from action a happening on state x. Now, consider the probability $P_a(x, y)$ for a given state $x \in S$ and an action $a \in A_x$,where A_x is the action space of that state x. Then $P_a(x, y)$ gives the probability of the state y occurring after action a has happened on x, and is defined as:

$$P_a(x,y) = \begin{cases} 0 & \text{if board in } y \text{ is not the result} \\ & \text{of action } a \text{ happening on board in } x \\ \\ \frac{1}{7} & \text{otherwise, since there are 7 possible} \\ & \text{pieces for the 1 possible board} \end{cases} \tag{4.2.5}$$

Thus, when calculating the expected cost to go of the next state after action a occurs on state x:

$$(\sum_{y \in S} P_a(x,y)\ \Phi(y)\,)r$$

we only need to consider the states $y \in Y_{a,x}$ where:

$$Y_{a,x} = \{\ (board, piece) \mid board\ =\ f(a,x);\ piece = 1..7\} \tag{4.2.6}$$

where

$$f(a,x) \in \Re^{20\times 10} \text{ gives the board when } a \text{ happens on } x$$

There are only seven possible states in $Y_{a,x}$. All seven states have the same board configuration and only vary at the shape of the falling piece. So, to redefine our problem (4.2.1) using the set $Y_{a,x}$ defined in (4.2.6) and the findings of (4.2.3) and (4.2.4) we get:

$$\begin{aligned} \max_r \quad & \left(\sum_{x \in \bar{X}} \Phi(x) \right) r \\ s.t. \quad & (\Phi(x) - \alpha \sum_{y \in Y_{ax}} P_a(x,y)\ \Phi(y)\,)\ r \le g_a(x) \\ & \forall (x,a) \in \bar{X} \end{aligned} \tag{4.2.7}$$

But, our basis functions Φ do not consider the falling piece of a state x, only the board configuration of that x. Since there is only one board configuration in set Y_{ax}, we have:

$$\Phi(y) = c,\ c \in \Re^K; \forall y \in Y_{ax}$$

where c is some constant. Using this with the definition of $P_a(x, y)$ (4.2.5) and noting that $|Y| = 7$, we have

$$\sum_{y \in Y_{ax}} P_a(x, y)\Phi(y) = \Phi(\widetilde{y})$$

where $\widetilde{y}$ any element of Y_{ax}. So the problem to solve (4.2.7) becomes:

$$\begin{aligned} \max_r \quad & \left(\sum_{x \in \bar{X}} \Phi(x) \right) r \\ s.t. \quad & (\Phi(x) - \alpha\Phi(\widetilde{y}))r \leq g_a(x) \\ & \forall (x, a) \in \bar{X};\ \widetilde{y} \text{ any element of } Y_{ax} \end{aligned} \tag{4.2.8}$$

Re-writing (4.2.8) to the standard LP form;

$$\begin{aligned} \min_r \quad & c\, r \\ s.t \quad & \\ & Ar \leq b \\ & \text{r is unbounded} \\ where: & \\ c = \quad & -\sum_{x \in \bar{X}} \Phi(x) \\ A = \quad & [Phi(x) - \alpha\Phi(\widetilde{y})] \\ b = \quad & [g_a(x)] \\ & \forall (x, a) \in \bar{X};\ \widetilde{y} \text{ any element of } Y_{ax} \end{aligned} \tag{4.2.9}$$

4.3 Implementation

The system was modularized and implemented in Matlab as three separate subsystems:

(a) The simulator/sampler.
(b) The constraint/coefficient generator
(c) The optimization solver

Matlab was chosen as the implementation language due to its simplicity and ability to store large number of data as binary files (.mat files). Matlab also has the advantage of interfacing well with optimization toolkits that can handle large number of constraints.

Simulator/ Sampler

The simulator plays the Tetris game over a number of configurable times. It plays according to the greedy policy generated by the basis functions Φ and a specified weight vector r which is passed in as input. The simulator includes a sampling mechanism that logs the following:

1. The game state (board and falling piece): x
2. the action chosen by the current policy: a
3. the resulting board when the action a is applied to state x: $\widetilde{y}$.

at a configurable sampling interval of M moves. All three pieces of data are stored into a file, The M moves used as the sampling interval may overlap between successive games. That is the sampling counter is not reset at the beginning of each game, but continues from the previous game. In order to speed up the sampling process, we use a bash script that starts the simulator on 5 different machines. The simulator runs concurrently on the machines, with each machine storing its samples in a different .mat file. Once all simulators are finished, the files are combined together into a single file as required by the constraint generator. This provides the advantage of using samples derived from different policies, as required by the guarded sampling extension of the BRLP (see section 3.4).

Constraint/Coefficient Generator

The constraint generator takes as input a single mat file that contains all states and chosen actions, recorded as a table. The generator proceeds to create the necessary constraints along with the objective coefficients, as required by the optimizer. The constraints and objective coefficients are then stored into a mat file. The generator creates:

1. objective coefficient vector $c = -\sum_{x \in \bar{X}} \Phi(x)$
2. the constrain coefficient matrix $A = [Phi(x) - \alpha\Phi(\widetilde{y})]$
3. constraint rhs vector $b = [g_a(x)]$

For the constraint elements (matrix A and vector b), the generator can be configured to work in two different ways:

(a) It can create one constraint for each possible action, for the state x in the sample. The generator will extract the sample state x and ignore the

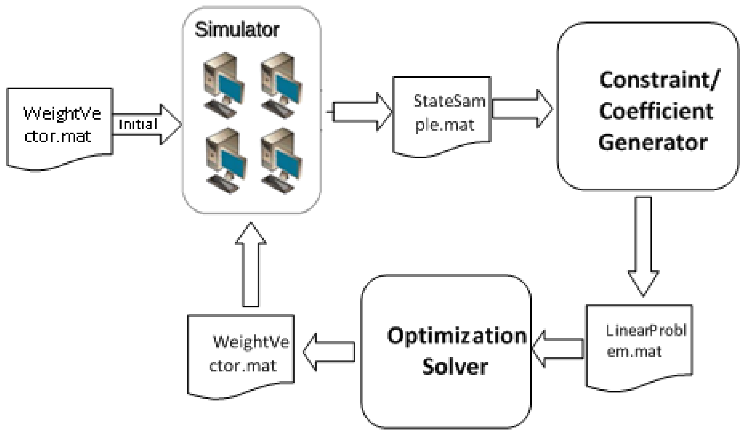

Figure 4.3: *Data flow for the Tetris system. The separate components communicate with each other via .mat files*

corresponding sampled action. It will retrieve the action space A_x, that is, the set of admissible actions for that particular state. It will then create one constraint for every action in A_x. The available actions for a particular state depend on the falling piece, and there are on average 23 combinations (see Table 4.1.2) for rotating and placing the pieces on the board.

(b) It can create one constraint for every sampled state x and the corresponding action a, which was chosen by the policy running during the simulation. That is, it will only create a constraint for the state-action pair that exists in the sample.

Whilst the first configuration is the which was successfully used [6], it has the disadvantage that it limits the sample space. This is due to the fact that there is a fixed number of constraints that the optimization toolkit can handle. Generating one constraint per possible action for the given state, limits the amount of sample states that can be used. The amount of state samples in the Tetris are reduced by a factor of 23, as there are 23 possible actions on average for each state (see Table 4.1.2.

That is, for 2,000,000 constraints we can only sample $2000000/23 \approx 90000$ states. However, this limitation is a lot more significant when it comes to problems with a lot more actions available per state, as is the Routing problem (chapter 5). Thus, we need to examine how the BRLP method works when the second configuration is used, and see whether we can use the second technique if the first one does not succeed.

Optimization Solver

The optimization solver takes as input the .mat file with the constraints and objective coefficients and solves the standard LP (4.2.9). The Mosek optimization engine was used to solve the LP. The Mosek engine handles 1,000,000 constraints on a machine that runs on 32-bit addressing space. Mosek has the advantage that it interfaces directly with Matlab and there is no need for the problem to be passed in a human readable equation format in a text file. The amount of constraints will render such a text file unmanageable in terms of size, and this is the main reason why attempts to optimization engines with this kind of interface, such as CPLEX, failed.

#	Options	Details
1	Sampling Interval M	Every M moves, we take a sample of the state, The moves may overlap between games. i.e. If a game terminates and there are 60 moves made since the last sample,we will take the next sample at 30 moves into the new game.
2	# of Constraints generated per state	We can either generate 1 constraint per sampled state and action (1 per (x, a)), where the sampled action is the action chosen by the current policy; or for each sampled state, generate 1 constraint constraint for every possible action (A_x per x).
3	New samples vs old samples in the sample set	To allow for guarded sampling, the sample may be constructed from a combination of new states,sampled according to the current policy, and states taken from the previous sample. The ratio of new states vs the states taken from the previous sample may be adjusted. In any case, the size of the sample is always sufficient to create the full number of constraints, that Mosek can handle.

Table 4.2: *The configurable options of the Tetris implementation.*

CHAPTER 5

Routing Under Uncertainty

The Routing problem under investigation is motivated by the problems faced with transmitting data through overlay networks. Overlay networks are networks that have high uncertainty with regards to the bandwidth of their links. Our objective is to repeatedly transmit a maximum amount of data through a network whose links have uncertain bandwidths. Some sort of bandwidth measurement is allowed but only on a limited number of links per unit time, right before transmission. The uncertainty about the bandwidths increases with the time since the last measurement.

5.1 Problem Specification

5.1.1 Description

In this section we are dealing with the transmission of data through a network. The network is represented as an undirected graph $G = (V, E)$ with a set of nodes V and a set of bi-directional links(edges) E. The graph may be fully connected, but there is at most one link connecting two nodes. The graph is undirected in order to represent the ability of data to flow in either direction between two nodes. For this problem, there is a source node S and a sink(destination) node T.We need to transmit data from S to T over a number of times. There is a finite set of available paths from S to T, that pass from different nodes. The paths may share links between them. The paths do not contain any loops as it makes no sense to go through them. According to basic graph theory, there are at most $n(n-1)/2$ edges in a graph with n nodes.

Each link connecting two nodes is associated with a bandwidth. That bandwidth is the maximum load of data that the link can carry and the system must never put more strain on a link than that value. The link's bandwidth is not certain and changes with time. A central controller can measure only a limited number of links (say 1 for simplicity) at each time step, but it can be any link

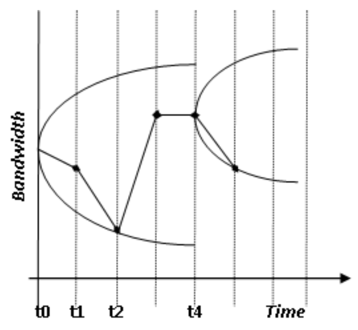

Figure 5.1: *the bandwidth behavior of a link. Uncertainty grows with time since last measurement.However, the minimum and maximum bandwidth can be guaranteed with a specific confidence probability.*

in the network. A link's bandwidth becomes more and more uncertain as time since the last measurement increases. However, the value that the bandwidth may have obeys a known probability distribution that is conditioned on both the time since last measurement as well as the last measured value. That is to say, the variance of the link's bandwidth grows with time, and the expected value of a measurement depends on the value of the last measurement. The lowest and highest values that the bandwidth of a link can take can be established using that distribution. As with the distribution, these bounds also depend on both the time since the last measurement was made as well as the value of the last measurement. Graph 5.1.1 shows the bandwidth of a single link. At each discrete time step, a link has bandwidth indicated by the dots. However, this bandwidth is not observable by the system unless the link is measured, as it happen at t0 and t4. Once a link is measured the system can establish a lower and upper bound on the possible values of the link's bandwidth, shown by the curves. The bounds grow wider as time since measurement increases. In order to use a specific link, we must transmit data at a bandwidth no greater than the lower bound of the link. Given the lower bound of each link in a path, one can transmit data from source to sink using that path only if he/she transmits at the bandwidth that ensures that it will not exceed the lower bound of any link in the path. That is, to use that path we must transmit at the lowest lower bound of the links in the path.

The system repeats the following sequence:

1. Choose a link to measure and measure it
2. Update the measured link, and the bounds on all the other links according to the time since last measurement.
3. Go through all the paths and for each path, establish the path's bandwidth

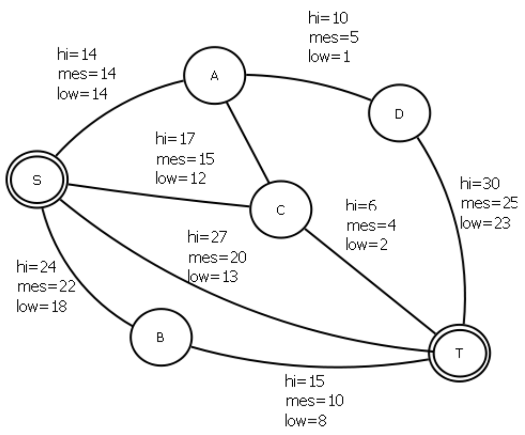

Figure 5.2: *The figure shows a network from a controller's perspective, with source node S and destination node T. There are 5 paths available from S to T. Each link has associated with it a measured bandwidth, and a time since last measurement(not shown). From the two, we can establish the lowest and upper bounds of the link's bandwidth. In order to use a particular link, we must transmit at a bandwidth not greater than the lower bound. If we want to use the path S-A-D-T we must transmit at a bandwidth of 1, which is the low bound of link A to D. In this scenario, we can see that the link that was measured before transmission is S-A , because the bounds are equal to the measured value.*

b_p that guarantees to be below or equal to the lower bound of all links within that path

4. Select the path with the highest b_p and transmit at that bandwidth
5. Increase the time and repeat

The decision that the controller needs to make, is which link to measure, in order to maximize the bandwidth with which we transmit.

5.1.2 Formulation as a MDP

To formulate the Routing problem as an MDP we need to specify:

(a) The State space definition S
(b) The Action space definition A_x for each state $x \in S$,
(c) The probability $P_a(x, y)$ that action a on state x will lead to state y
(d) The expected immediate cost $g_a(x)$ of action a happening on state x

State Definition
At any particular time each link in the graph has associated with it the following:

1. the measured bandwidth
2. time since last measurement
3. the probability distribution over the link's bandwidth

From the above we can derive the minimum and maximum possible bandwidths that the link may have. In order to keep the state space finite, we need to discretize both the bandwidth space and the time space. Also, for simplicity we assume that the probability distribution does not change with time, meaning that we can move it out of the state and make it a part of the system's structure. However, this distribution is conditioned on the measured bandwidth and the time since last measurement. Even though the distribution is part of the structure, it does depend on the present state. The MDP state of the system $G = (V, E)$ is defined as the set:

$$S = \{\ (b_e, t_e) | e \in E\ \} \tag{5.1.1}$$

$$\begin{aligned} where \quad & \\ & t_e \text{ time since last measurement of link } e \\ & b_e \text{ last measured bandwidth level of link } e \\ & t_e \in T;\ b_e \in B;\ B = 1 \dots M;\ T \subset \mathbb{N}_0 \end{aligned} \tag{5.1.2}$$

Note that the bandwidth is discretized as integer levels and has a maximum and minimum value. That is, at any given point in time, no link can have a possible bandwidth value that is more than level M and less than level 1. For simplicity we assume that this is the set of bandwidths that any link can take, i.e. this set does not depend neither on the specific link nor on the system running time. Time is also discretized. Since time t_e is time since last measurement, it is an integer between 0 (for the single link that is measured now) and $+\infty$.

An important observation is that the state space is still infinite, even after we discretized the time and bandwidth. This is because time is taken from an infinite discrete set. However, this is not an issue, because, as we will see later on, given a state x, the set $Y_{a,x}$ containing all the possible states that may follow from action a happening on state x is a finite set.

Action
The rooting problem is not about choosing the optimal route, but the link that we need to measure. Given the state of the system, the optimal route is chosen

analytically as the one that maximizes the bandwidth (see the immediate cost subsection). Thus for each state, the action set is always the same, and its identical to the set of links E. That is,

$$A_x = E, \forall x \in S$$

In other words, the action is the choice of the single link to measure, and at any time step we can measure any link we want.

Next state Set
Given an action a, and a state $x = \{(b_e, t_e) | \; e \in E\}$. Our next state y is:

$$y = \{(b_e, t_e + 1) \mid e \neq a; \; e \in E; \; (b_e, t_e) \in x\} \bigcup \{(b'_a, 0)\} \tag{5.1.3}$$

That is, if we choose to measure link a in the current state x, in the next state y, all other links will have the same measured bandwidth b_e as in the current state x. However, the time since last measurement will increase by one, since we are not measuring them at this stage. For our chosen link a, the time since last measurement is set to 0, since we measure the link at the present time right before transmission. The new measured bandwidth b'_a of link a will be the outcome of the measurement. This outcome is according to the link's probability distribution, conditioned on the previous measurement b_a and the time since last measurement t_a. The new bandwidth b'_a is defined as a random variable where:

$$b'_a \sim F_a(b_a, t_a)$$

where F_a is the probability distribution for the bandwidth of the chosen link, and it depends on the link's present state.

The state of the system evolves in a deterministic matter, apart from the chosen link's measured bandwidth. For a given action a happening on state x, the number of different states y that may follow is equal to the number of possible values that b'_a may take. However, the number of values that b'_a may take is bounded, since there is a finite number of possible bandwidth levels equal to M. So there are at most M possible states that may follow after a certain action a has taken place on current state x. The set $Y_{a,x}$ that contains all possible states y that may follow after a happens on x, is defined as:

$$Y_{a,x} = \{\{(b_e, t_e + 1) \mid e \neq a, \; e \in E; (b_e, t_e) \in x\} \bigcup \{(b'_a, 0)\} \mid b'_a = 1 \ldots M\} \tag{5.1.4}$$

This is an important result. It allows for the Dynamic Programming methodology to be applied, even though the state space is not finite.

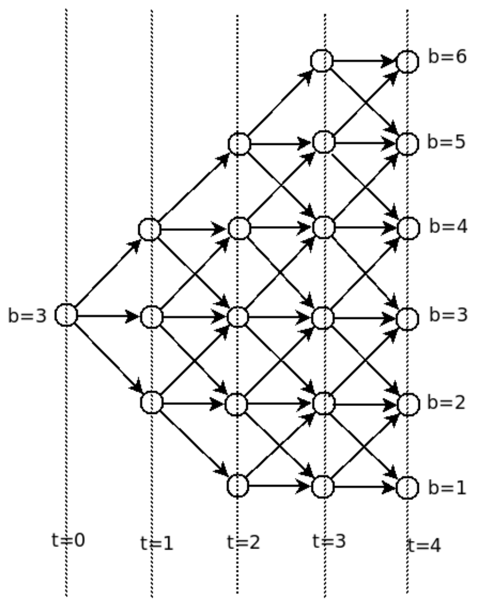

Figure 5.3: *Lattice demonstrating the possible bandwidths at each time. The figure shows the system with maximum bandwidth level equal to $b = 6$, that is $M = 6$. The available bandwidths at are taken from the finite set $\{1, 2, \ldots, 6\}$. The measured bandwidth $b_e = 3$ and the time since measurement $t_e = 4$.*

State probabilities $P_a(x, y)$
For any state $y \in S$, we can define the probability $P_a(x, y)$ of the specific state y following after after action a happens on x. First, note that according to the set definition $Y_{a,x}$ in (5.1.4), the probability $P_a(x, y)$ of any state $y' \in S$; $y' \notin Y_{a,x}$ is equal to 0. This is to say, that the set $Y_{a,x}$ as defined in (5.1.4) contains all the possible states that may follow after a happens on x, and any state that is not contained in that set has 0 probability of occurring.

Now consider a state $y \in Y_{a,x}$. Let $(b'_a, t'_a) \in y$. That is, for state y the bandwidth of the link a that is chosen to be measured is found to be b'_a. The probability $P_a(x, y)$ of that state y following after action a happens on x, is equal to the probability that link a is found to have a bandwidth b'_a after measurement. This is because all other elements in state y are deterministic, and defined according to (5.1.3). The only element that differentiates state y from any other state in the set $Y_{a,x}$ is the outcome of the measurement b'_a. Thus the probability $P_a(x, y)$ that y will follow x after action a happens on it, is equal to the probability that the bandwidth of link a is found to be b'_a. But the probability of link a having a certain value is governed by the link's conditional probability distribution $F_a(b_a, t_a)$ where b_a and t_a is the current state's information about the link. Thus to define $P_a(x, y)$ we need to define the probability mass function $f_{a,b_a,t_a}(\cdot)$

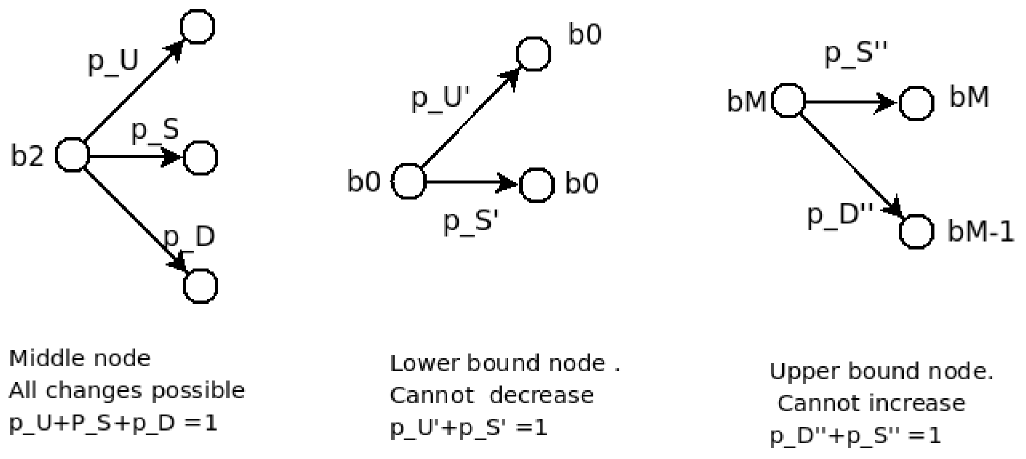

Figure 5.4: *The three different types of nodes,center node, lower bound node, and upper bound node, along with their possible changes and transitional probabilities associated with each change*

.

for that link's distribution $F_a(b_a, t_a)$. The probability mass function gives us the probability with which the link's bandwidth will have a specific value. Once we have defined the mass function, we can define the probability $P_a(x, y)$ for all possible states as follows:

$$P_a(x,y) = \begin{cases} 0 & \text{if } y \notin Y_{a,x} \\ \\ f_{a,b_a,t_a}(b'_a) & \text{where } (b'_a, 0) \in y;\ (b_a, t_a) \in x \end{cases} \tag{5.1.5}$$

For simplicity, we formulate all distributions F_a using a finite trinomial lattice with upper and lower limits $b = 1$ and $b = M$ respectively.

Figure 5.3 shows the possible changes in a link's bandwidth. The bandwidth takes discrete values, from the set $B = \{1, 2, \ldots, 6\}$. Thus, the diagram is for measured bandwidth $b_e = 3$, and time since last measurement $t_e = 4$. Using a trionomial lattice we observe the following:

(a) There are 3 different ways a bandwidth can 'move' at each state, either go up, down , or remain the same. This happens with transitional probabilities p_u, p_d and p_s. However, not all the changes are available at all states.This is because there are upper and lower limits, which restrict the bandwidth from going up or down respectively, at the next state.This is demonstrated in figure 5.4. Thus, there are three sets of transitional probabilities $([p_u, p_d, p_s][p'_u, p'_s],[p''_d, p''_s])$.The assumption that the probability dis-

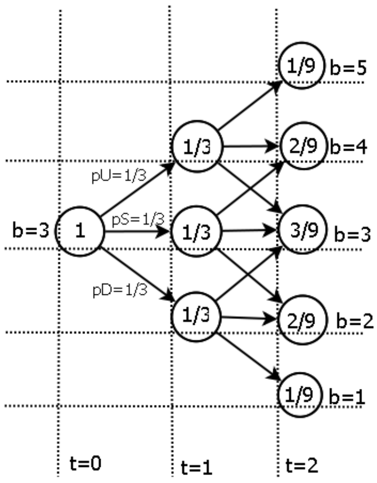

Figure 5.5: *The probability for various bandwidth levels of a specific link, when the last measured bandwidth level was found to be* $b_a = 3$. *This link has transitional probabilities* $p_U = p_D = p_S = \frac{1}{3}$ *which remains constant with time. The probability for each bandwidth level, for the different time intervals is shown in the circles. Note that at* $t_e = 1$, *the possible bandwidth levels are from the set* $\{2, 3, 4\}$, *whilst for* $t_e = 2$ *the set grows to* $\{1, 2, 3, 4, 5\}$. *Any bandwidth level not shown, or not connected to the lattice at a specific time since last measurement, is not possible and has probability of occurring 0.*

tribution over a link's bandwidth does not change with time, translates into these probabilities being constant over time.

(b) The lower and upper bounds increase, until they include all available bandwidth levels (1 to M). From a specific time since last measurement interval, the probability distributions associated with the possible bandwidth levels becomes stable, and does not change as time since last measurement increases. We call this time interval T_{stable}, and we note that the probability mass function after T_{stable} remains the same.

(c) It is easy to see that the upper and lower bounds of possible bandwidths for a given link with b_e at time since measurement t_e, are given by, upper: $minb_e + t_e, M)$ where M the maximum allowable bandwidth level, and lower: $max(b_m - t_e, 1)$, where 1 the minimum allowable bandwidth level.

We need one trinomial lattice per bandwidth level per link. Since this is a finite trinomial lattice, we cannot define the probability mass function using an analytical method. However, we can populate the lattice iteratively using the known transitional probabilities. Let b_t denote the bandwidth at time since last measurement equal to t. Assume that the bandwidth was last measured and found to be at level n. That is,the initial node in our lattice will be $b_0 = n$, and is assigned probability 1 ($P[b_0 = n] = 1$). Starting with the initial node we calculate the probabilities of the next three nodes ($b_1 = n - 1, b_1 = n, b_1 = n + 1$)as :

$$\begin{aligned} P[b_1 = n - 1] &= P[b_0 = n] * p_d \\ P[b_1 = n] &= P[b_0 = n] * p_s \\ P[b_1 = n + 1] &= P[b_0 = n] * p_u \end{aligned}$$

where p_d, p_s, p_u are the three known transitional probabilities. This can be generalized for any node as:

$$\begin{aligned} P[b_t = n] = {} & P[b_{t-1} = n + 1] * p_d \\ & + P[b_{t-1} = n] * p_s + P[b_{t-1} = n - 1] * p_u \end{aligned}$$

That is, the probability of the node with bandwidth at level n and time since last measurement t is the probability of the previous node being at the immediately higher level times the transitional probability of a drop in bandwidth, plus the probability of the previous node being at the same level times the transitional probability of remaining at that level, plus the probability of the previous node being at the immediately lower level times the transitional probability of a gain in bandwidth during the transition(Figure 5.1.2 shows an example where the lattice is populated). We can easily adjust this equation to account for the absolute bounds $b = 1$ and $b = M$. This is best done if we index the probabilities according to the bandwidth level, and set the transitional probabilities for going beyond the maximum and below the minimum to 0. Thus the generalization becomes:

$$\begin{aligned} P[b_t = n] = {} & P[b_{t-1} = n + 1] * p_{d,n+1} \\ & + P[b_{t-1} = n] * p_{s,n} + P[b_{t-1} = n - 1] * p_u n - 1 \end{aligned}$$

Using the three sets of transitional probabilities (($[p_u, p_d, p_s]$,$[p'_u, p'_s]$ and $[p''_d, p''_s]$), we define the indexed probabilities as :

$$p_{d,n} = \begin{cases} p_d & n = 2 \dots M-1 \\ 0 & n = 1 \\ p''_d & n = M \end{cases}$$
$$p_{s,n} = \begin{cases} p_s & n = 2 \dots M-1 \\ p'_s & n = 1 \\ p''_s & n = M \end{cases}$$
$$p_{u,n} = \begin{cases} p_u & n = 2 \dots M-1 \\ p'_u & n = 1 \\ 0 & n = M \end{cases}$$

We need to expand the lattice until a stable distribution is reached. Then the probability mass function is derived by indexing the correct lattice for the given link and measured bandwidth level, and retrieving the probability of each bandwidth level for that specific time since last measurement from the lattice.

Immediate Action Cost $g_a(x)$
At each time step, data are transmitted from the source node S to sink node T, at the maximum safest bandwidth level. The maximum safest level is the one that we can guarantee that no link along the chosen path will have a lower bandwidth than that. Since a link bandwidth is uncertain and obeys a finite trinomial lattice tree, as shown in figure 5.3, we define the bounds of the bandwidth that a link e can take using the previous measurement b_e and the time since last measurement t_e as:

- .The lower bound of the bandwidth $b_{lo,e} = \max(b_m - t_e, 1)$
- .The upper bound of the bandwidth $b_{hi,e} = \min(b_m + t_e, M)$.

We assume that there is a fixed set of known paths P from source node S to sink node T, and that for any path $p \in P$; $p = \{e_s, ..., e_t\}, e_i \in E \, \forall i$, the bandwidth of the path p at time t when the system is at state x is :

$$\begin{aligned} b_p &= \min_e \{ b_{lo,e} \,|\, \forall e \in p \} \qquad (5.1.6) \\ \Leftrightarrow b_p &= \min_e \{ \min\{b_e - t_e, M\} \,|\, \forall e \in p;\ (b_e, t_e) \in x \} \end{aligned}$$

That is, the path bandwidth is the lowest lower bound over all the edges in that path. Choosing to send data over that path at that bandwidth, is guaranteed with probability 1 to arrive at the sink, without exceeding the bandwidth level of any link along the way.

Our chosen path p^* is the one that solves :

$$\begin{aligned} p^* &= \underset{p}{\operatorname{argmax}}\{b_p \,|\, \forall p \in P\} \\ \Leftrightarrow p^* &= \underset{p}{\operatorname{argmax}}\{\min_e\{\, b_{lo,e} \,|\, \forall e \in p;\ (b_e, t_e) \in x\} \mid \forall p \in P\} \end{aligned} \tag{5.1.7}$$

and the bandwidth with which we will transmit the data will be:

$$b_{p^*} = \min\{\, \min(b_e - t_e, M) |\, \forall e \in p^*\} \tag{5.1.8}$$

Our objective is to maximize that bandwidth by choosing the appropriate link to measure at each time. Thus, we are no longer considering an immediate cost, but an immediate profit. However we will maintain the same notation, so the immediate profit for action a is $g_a(x) = b_{p^*}$. Note that, unlike the Tetris game, this immediate profit is not really immediate. That is, we cannot at the time we choose our action, i.e. which link to measure, determine the immediate cost that our action will have. The profit is realized after we have performed the chosen action, that is, after we have measured the link and found its current bandwidth level. This profit does not play part in the greedy policy, and is only used to formulate the optimization problem that we need to solve:

$$\sum_{t=0}^{\infty} \{\alpha^t g_{a_t}(x_t)\}$$

. However, a way of having the greedy policy consider the immediate profit of an action, is to include the profit in the basis functions. Thus, at least the expected immediate profit will be considered by the policy when choosing an action.

5.1.3 Approximate Cost-to-Go for the Routing Problem

Given a graph $G = (V, E)$ with $|V|$ nodes and $|E|$ links, we introduce a set of basis functions that will be used in the parametric cost-to-go approximation as follows:

(a) $|E|$ basis functions, each mapping the state to the measured bandwidth level b_e for each of the links e

(b) $|E|$ basis functions, each mapping the state to the time since last measurement t_e for each of the links e

(c) $|E|$ basis functions, each mapping the state to the minimum possible bandwidth $b_{lo,e}$ for each of the links e

(d) $|E|$ basis functions, each mapping the state to the minimum possible bandwidth $b_{hi,e}$ for each of the links e

(e) 1 basis function that is equal to one at every state, serving as a constant offset

Thus we have a total of $|E| * 4 + 1$ basis functions. The number of basis functions and thus objective variables depends only on the number of links, and not the number of bandwidth levels. As discussed earlier, if we want to include the immediate action profit as defined in (5.1.8), or at least the expected value of it, in our greedy policy, we can include it in the basis functions, However, calculating the immediate profit for every possible outcome of every possible action is a very time consuming procedure, that delays the greedy policy significantly. Particular, consider a graph of n nodes and l links. If there are on average b possible outcomes for every action, then we need to solve (5.1.8) $b * l$ times. That is, b times for all possible outcomes for each action, multiplied by l which is the number of links and thus the number of possible actions. To solve (5.1.8) we need to traverse all paths in order to find the one with the largest path bandwidth, as defined in (5.1.6). This is a major bottleneck and a greedy policy that includes such a problem will introduce significant delays to the network when trying to choose the next action.

5.1.4 Performance Objectives

As this is a unique problem, there aren't any benchmarks that we can use to evaluate a policy's performance. However, we have devised a test that will allow as to measure the performance compared to some other logically derived policies . We run a simulation of the network and have various controllers acting on it. Each controller has its own state, which is the set of information about the network (i.e. measured bandwidth and time since last measurement for each of the links). Each controller follows its own policy in updating its state. At each time step, we record the transmission bandwidth achieved when using each controller's state. In other words, we derive the immediate profit g_a for each of the controllers. Thus, we can compare the average performance of the controllers running in the same environment over a sufficiently large period of times. The reason we can run different policies on the same simulation, is because a policy does not affect the actual network's evolution. Measuring a link will not force that link to take a certain bandwidth level. A policy's action only affects the information we know about the network. We can compare the policy derived by the BRLP method against the following policies:

(a) **A heuristic policy.** The policy is designed based on common sense. This is a greedy policy, that uses the same basis functions to approximate the cost-to-go as the policy on test. However, for this policy, we use a specially hand crafted weight vector r, which is logically designed so that:

- the direct link between source and sink is measured more often than those links that are only connected to either the source or the sink, and the latter links are measured more often than the links between the other nodes.
- links with higher uncertainty are more likely to be measured than links with low uncertainty.
- links with high expected bandwidth are more likely to be measured than links with low expected bandwidth.
- links that have not been measured for some time are more likely to be measured

This evidently, is the same weight vector that we use as the starting point for the BRLP method

(b) **A round robin policy.** This policy measures the links in a round robin fashion.

(c) **A perfect-information policy.** This policy is derived from the simulator. The policy tabs into the simulator data and maintains the current bandwidths of all the links. This policy represents the ability to measure all links at each time step. It has the optimal performance that can be achieved by the network.

The objective is for the derived policy to at least surpass the round robin policy, and hopefully the heuristic policy. Ideally, we would like the derived policy to closely match the performance of the perfect information policy.

5.2 Problem Formulation

The routing problem varies from the Tetris problem, for three reasons:

(a) The state space S infinite

(b) We are dealing with action profits, not costs. The objective now is to maximize the accumulated profit $g_a(x)$ over time.

(c) The immediate action profit is not realized until after the action, and is non-deterministic

These differences require a reformulation of both the RLP objective, as well as the greedy policy's definition. The main difference lies in the way we calculate the expected future cost, or in this case profit.

5.2.1 Expected Future Profit

Given a cost-to-go approximation $\widetilde{J}$, so far we have been calculating the expected future cost of action a happening on state x as (see section 2.2):

$$\sum_{y \in S} P_a(x,y)\widetilde{J}(y) \tag{5.2.9}$$

That is ,calculate the weighted average the cost-to-go over all states in the state space S, according to the probability $P_a(x,y)$ that those states may follow after a happens on x. Such a calculation is intractable for the Routing problem, because the state space S is now infinite. However, the set $Y_{a,x}$ that contains all possible states that may follow once action a happens on state x is finite, as shown earlier.

Thus, by noting that $P_a(x,y)$ is 0 (see (5.1.5)) for any state not in the set $Y_{a,x}$, we can reformulate the Expected Future Profit in a tractable equation by summing over the states in the finite set $Y_{a,x}$ rather the entire state space. So the expected future cost for action a on x is:

$$\sum_{y \in Y_{a,x}} P_a(x,y)\widetilde{J}(y) \tag{5.2.10}$$

5.2.2 RLP modification for the expected Profit-To-Go

The problems we have dealt with so far aimed at minimizing the total cost accumulated by running the system over time. The BLRP method discussed is set up to find the best cost-to-go approximation for each state, given the choice of basis functions. However, in this problem we need to maximize the total accumulated profit, so a profit-to-go approximation is needed now.

Remember, the cost-to-go for a state is defined as the minimum total expected cost that will be incurred when the system starts at that state and runs under the optimal policy. The cost-to-go is the solution to Bellman's equation (see section 2.2):

$$J^*(x) = \min a \in A_x \left\{ g_a(x) + \alpha \sum_{y \in S} P_a(x,y) J^*(y) \right\} \tag{5.2.11}$$

The exact version of the Linear Problem that solves the above equation is (see subsection 2.4.1 :

$$\begin{aligned}\max_{J} \quad & c'J \\ \text{s.t.} \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x,y)J(y) \geq J(x) \\ & \forall x \in S;\ \forall a \in A_x\end{aligned} \tag{5.2.12}$$

Similarly to the cost-to-go, the profit-to-to is defined to be equal with the maximum total expected profit that will be incurred, when the system starts from the specific state under the optimal policy. Thus, maintaining the same notation, the profit-to-go is defined as :

$$J^*(x) = \max_{a \in A_x} \left\{ g_a(x) + \alpha \sum_{y \in S} P_a(x,y)J^*(y) \right\} \tag{5.2.13}$$

The Linear program needs to be modified to return the expected profit-to-go as:

$$\begin{aligned}\min_{J} \quad & c'J \\ \text{s.t.} \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x,y)J(y) \leq J(x) \\ & \forall x \in S;\ \forall a \in A_x\end{aligned} \tag{5.2.14}$$

Any solution to the above problem is both optimal and feasible. Thus, by the constraint definition, any solution J will be greater than the left-hand side. The profit-to-go J can only decrease up to the point where it is greater than the left hand side. As a result, J will be equal to the largest allowable value of

$$g_a(x) + \alpha \sum_{y \in S} P_a(x,y)J(y)$$

which is what is required by the profit-to-go definition (5.2.13).

In order to bring the problem (5.2.14) to its RLP form, we modify it in the same manner as we did with its cost-to-go version(see chapter 3. Thus the RLP we need to solve now is:

$$\begin{aligned}\min_{r} \quad & c'\Phi r \\ \text{s.t.} \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x,y)(\Phi r)(y) \leq (\Phi r)(x) \\ & \forall (x,a) \in \overline{X}\end{aligned} \tag{5.2.15}$$

However, as with subsection 5.2.1, we need to replace the expected future profit term in the constraints

$$\sum_{y \in S} P_a(x, y)(\Phi r)(y)$$

with its tractable form

$$\sum_{y \in Y_{a,x}} P_a(x, y)(\Phi r)(y)$$

where $Y_{a,x}$ the set of next possible states as defined in (5.1.4). The final version of the RLP that is used by the BRLP method is:

$$\begin{aligned} \min_r \quad & c'\Phi r \\ \text{s.t.} \quad & g_a(x) + \alpha \sum_{y \in S} P_a(x, y)(\Phi r)(y) \leq (\Phi r)(x) \\ & \forall (x, a) \in \overline{X} \end{aligned} \tag{5.2.16}$$

5.2.3 Greedy policy

When dealing with cost, the greedy policy generated by a cost-to-go approximation $\widetilde{J} = \Phi r$ needs to choose the action that solves (see section 2.2:

$$u(x) = \arg\min_{a \in A_x} \left\{ g_a(x) + \alpha \sum_{y \in S} P_a(x, y) \widetilde{J}(y) \right\} \tag{5.2.17}$$

That is, the policy will choose the action that minimizes the sum of the expected future cost and the immediate action cost. Now that we are considering a profit instead of a cost, we need to redefine the greedy policy to choose the action that maximizes the sum of the immediate action profit and the expected future profit. Thus, the greedy policy is now defined as :

$$u(x) = \operatorname*{argmax}_{a \in A_x} \left\{ g_a(x) + \alpha \sum_{y \in Y_{a,x}} P_a(x, y) \widetilde{J}(y) \right\} \tag{5.2.18}$$

where $g_a(x)$ is now the immediate action profit and $\widetilde{J}$ is now the parametric approximation of the profit-to-go function. We again replace the state space S with the set of the next possible states $Y_{a,x}$, in order to calculate the expected future profit.

The immediate profit of an action is not readily available when choosing the action, but is realized after the action has been chosen (see subsection subsection 5.1.2). Thus, the immediate profit has no weighting when choosing an action. As a result, the greedy policy becomes:

$$u(x) = \operatorname*{argmax}_{a \in A_x} \{ \sum_{y \in Y_{a,x}} P_a(x, y) \widetilde{J}(y) \} \tag{5.2.19}$$

Finally, the set of admissible actions is the same at each state, and is identical to the set of links (see 5.1.2. Thus, we can replace A_x with E, where E is the set of links. The final greedy policy to be used in the rooting problem is :

$$u(x) = \operatorname*{argmax}_{a \in E} \{ \sum_{y \in Y_{a,x}} P_a(x, y) \widetilde{J}(y) \} \tag{5.2.20}$$

5.3 Implementation

The system was implemented in Matlab. Mosek was again used as the LP solver. This problem involves a continuously evolving network, so there is no scope for concurrent simulation, as we did with the Tetris (see section 4.3). The system was implemented as a self-contained program, in order to automate a BLRP iteration. The program is designed to run without any user interaction and perform a configurable number of BLRP iterations before terminating. Each iteration consists of :

1. Simulation/Sampling phase using current policy
2. Constraint and objective coefficient generation using the recorded sample
3. Solution of the linear program formulated in order to derive the policy to use in the next iteration

The program is designed to log all results at the end of each iteration, such as:

(a) the bandwidths achieved by the policy used in each iteration, as those bandwidths were observed in the simulation process,

(b) the weight vectors used in the each of the policies and the sample sets that were constructed in each iteration.

The program also backups regularly its entire internal state in a .mat file. As the program is left to run unattended as a background process, there is the possibility that the host computer crashes or reboots before completion. The backup points allow as to restore software execution, without having to start all over again.

5.3.1 Network Representation

The software needs to simulate the network and evolve the bandwidths of each link according to the link's pre-specified probability distributions. The links follow a trinomial lattice based distribution, and the simulator only needs to store the three sets of transitional probabilities shown in figure 5.4 for each of the links. There are two components that form the network's structure:

(a) The nodes and the links between them. The nodes and links are represented as a two dimensional array. If there are $|V|$ nodes, then the entire network is represented as an $|V| \times |V|$ symmetric matrix. If there is a link from node i to node j, then the entry at row i and column j is set to 1, and so is the entry at row j and column i. The rest of the entries are marked as 0, meaning that we can store the matrix in the more memory efficient sparse representation that Matlab provides. The number of nonzero elements in the matrix will be equal to $2|E|$ where $|E|$ is the number of links.

(b) The 3 sets (8 distinct probabilities in total) of probabilities that are associated with each link. In order to store these probabilities, each link is associated with a specific id. Thus, the above matrix is modified, so that if there is a link between node i and node j, instead of setting the ij and ji entries of the matrix, we set them to the link's specific id. We then use that id to index the row in separate table, that holds the specific probabilities for that link. There are 3 sets of 8 probabilities associated with each link, but we only need to store 5 distinct probabilities, as the other 3 can be derived by knowing that each of the three sets sums up to 1.

Overall, to represent the network of $|V|$ nodes and $|E|$ links we need an $|V| \times |V|$ matrix to hold the network topology and an $|E| \times (5 + 1)$ matrix to hold the probabilities.

5.3.2 Simulator/Sampler

The simulator needs to evolve the system at each time step, by progressing the bandwidth of each link according to the link's transitional probabilities, as they are stored in the network structure. In order to do so, the simulator needs to store the actual bandwidth of each of the links at each time step. This can be stored as an entry to the table that holds the probabilities associated with each link. So the matrix now needs to hold the 5 probabilities plus another entry which is the current bandwidth, for each of the links. The sampler needs to derive the action profit g_a for the sampling purposes. In order to do so, the sampler needs to go through all paths, and find the bandwidth of each path as defined in (5.1.6). To do this, the sampler needs to hold a list of paths, representing each path as an array of the link ids that the path consists of. This list is created on start up, using the network structure and a breadth-first search. The sampler also needs access to the MDP state that the policy maintains. Even though finding the best path along with the transmission bandwidth is implemented as part of the sampler, in a real life system this would be part of the controller, along with the greedy policy.

5.3.3 Greedy Policy Implementation

As discussed in subsection 5.2.3, the greedy policy of the routing problem only needs to consider the expected future profit given the choice of a particular action, and ignore the immediate action profit. In order to calculate the expected profit

$$\sum_{y \in Y} P_a(x, y)\Phi(y)r$$

, the greedy policy needs to:

(a) Generate the set of next states Y for each of the links
(b) Retrieve the probability $P_a(x, y)$ of a state y occurring after action a happens on state x
(c) Apply the basis functions to any state for every action and possible outcome

In order to generate the next possible state, the policy implementation needs to maintain the MDP state, as well as the set of trilattices (one for every bandwidth level) that are associated with each of the links.

Generate Set of next states
The policy stores the MDP state , as an $|E| \times 2$ matrix, where $|E|$ is the number of links. The row number is the specific index of that table, as that index was assigned in the network representation. Each row holds the measured bandwidth and time since last measurement for that link. Also, to save computation time, the table has another 2 entries on each row, which hold the minimum and maximum bandwidths, for the specific time since last measurement.We can derive the next state, both in the case that we have actually made a decision to measure a specific link, but also in the case that we want to create the set of next possible states, to be used by the greedy policy. Regardless of the situation, for all links apart from the one considered/chosen, we do the following:

(a) increment the time since last measurement by 1

(b) update the bounds according to:

- $.b_{lo,e} = \max(b_m - t_e, 1), \ (b_m, t_e) \in x$
- $.b_{hi,e} = \min(b_m + t_e, M), \ (b_m, t_e) \in x.$

(c) the measured bandwidth $b_m es$ remains the same

For the considered/chosen link, we set the time since measurement $t_m es$ to 0, and then we set the measured bandwidth $b_m es$, along with the $b_l o$ and $b_h i$ to:

(a) either the actual bandwidth as recorded by the simulator, in case this is the link that has been chosen to be measured

(b) the bandwidth as proposed by a potential next sate, in case we want to create the set of possible next states $Y_{a,x}$ if this link is considered. This will be used by the greedy policy to estimate the future expected profit-to-go if this link is chosen.

Calculating $P_a(x, y)$
The set of trilattices is generated using a separate Matlab program. The program will take as argument the number of bandwidth levels M and the number of paths $|P|$. It will then proceed to generate the three sets of transitional probabilities at random for each of the p links. Using the probabilities, it will populate the lattices, which are represented as a 2 dimensional array (1dimension are the bandwidths, the other is the time step). Thus, overall ,the lattices are stored as four dimensional arrays, where the first dimension, refers to the link, the second dimension, to each possible starting bandwidth, and the last two dimensions to the lattice. It will store the four dimensional table in a mat file, that is passed in the main program. From the trilattice, we can derive the probability $P_a(x, y)$ for any x, y and a. Let $(b_a, t_a) \in x$ be the measurement bandwidth and time since last measurement of link a in the current state. Let $(b'_a, 0) \in y$ be the same data in the proposed next state y. To get $P_a(x, y)$ we retrieve the set of lattices for link a, from that set we retrieve the lattice that has as a measured bandwidth b_a, and from there we index the value at the cell with bandwidth b'_a and time since measurement t_a.

Apply the basis functions
The greedy policy solves (5.2.20) by calculating the expected future cost for every possible action. The greedy policy works as follows:

```
method GreedyPolicy(state x) returns optimalAction
 CurrentState = x
  Set OptimalProfit = 0
  for every link a
    //calculate expected future profit
    set FutureProfit = 0
    [blow,bhi] = getRangeOfPossibleBandwidths(x,a)
    for bmes = blow up to bhi
      nextStateY = deriveNextState(x,bmes,a)
      BasisVector = phi(y)
      ProfitToGo = dotProduct(BasisVector,r)
      FutureProfit = FutureProfit + prob(x,y,a)*ProfitToGo
    end forloop
    //check if maximum
    if OptimalProfit< FutureProfit
      OptimalProfit = FutureProfit
     optimalAction = a
    endif
  endloop
  return optimalAction


method deriveNextState(state x,bandwidth bmes, link a)
    returns state y
 // this method, takes in the current state x, and
//the suggested measured bandwidth for the chosen link,
//it returns the next state, for that bandwidth

method getRangeOfPossibleBandwidths(x,a) returns [blo,bhi]
  //the method gets the current state x and the chosen link a
  // and returns the two bounds for that link, as they are
  //recorded in the state x


  method phi(state y) returns vector
  // the method applies the basis functions on the
  //state that is passed in as parameter
```

Thus the greedy policy needs to derive Φr for all possible outcomes of an action, for all possible actions. Given an action and an outcome, there are $4 * |E|$ basis that functions that we need to apply for that specific outcome state. That is, there are 4 basis functions for each of the links, that extract the measured bandwidth, the time since last measurement, the lower and the upper bounds. Remember, the number of possible actions is the number of links $|E|$. If there are on average n possible next states per action, then we need to apply the basis functions a total of $|E| * n$ times . Thus, we will need to do

$$|E| * n * 4 * |E| = 4n|E|^2$$

operations when we apply the greedy policy, in each iteration. As implemented above, the true work happens in the 'deriveNextState' method. The method calculates the new entries of all links, including the link that is chosen to be measured. The 'phi' method simply extracts the information already calculated in the 'deriveNextState' method, in a vector form. No calculation is done in the 'phi' method.

However, we note that :

(a) Given the chosen action, each possible outcome state only varies in 1 link from any other outcome. All other links have the same values. Namely, if we choose as action link a, then for any two outcomes states $y, y' \in Y_{a,x}$ where x is the present state, y will vary from y' only in the measured probability of link a. All other entries in the two states will be the same

(b) Any outcome of an action, will vary only in 2 links from any outcome of another action. Thus, if we take action a and a', then states y and y' resulting from a and a' respectively, will only vary in the entries for link a and a'.

Thus, we can derive an optimized implementation of the greedy policy as follows:

```
method GreedyPolicy(state x) returns optimalAction
 CurrentState = x
  set OptimalProfit = 0
  // derive next state if no action takes place
  yGen = genericNextState(x)
  for every link a
   set FutureProfit = 0
   [blow, bhi] = getRangeOfPossibleBandwidths(x,a)
   for bmes = blow up to bhi
     CurrentState= adjustState(yGen,bmes,a)
      BasisVector = phi(y)
     ProfitToGo = dotProduct(BasisVector , r)
    FutureProfit = FutureProfit + prob(x,y,a)*ProfitToGo
    end forloop
   //check if maximum
   if OptimalProfit< FutureProfit
    OptimalProfit = FutureProfit
     optimalAction = a
   endif
  endloop
  return optimalAction
```

```
method adjustState(state yGen,bandwidth bmes, link a)
 returns the specific next state
 // this method, takes in the generic next state x, and
//the suggested measured bandwidth for the chosen link,
//it returns the next state, for that bandwidth, by adjusting
//the selected link in the generic state appropriately


method genericNextState(x) returns a generic next state
// the method takes a present state, and returns
//the next state when no action is applied to it

method getRangeOfPossibleBandwidths(x,a) returns [blo,bhi]
//the method gets the current state x and the chosen link a
// and returns the two bounds for that link, as they
//are recorded in the state x

method phi(state y) returns vector
  // the method applies the basis functions on
  //the state that is passed in as parameter
```

The difference lie in two methods, the 'genericNextState' method and the 'adjustState' method. Thus the new implementation, only needs to derive an entire state one time, by calling the 'genericNextState' method which takes $4|E|$ operations. It then proceeds to modify the generic state $n * |E|$ times, by calling the 'adjustState' operation. That is, one time per possible outcome per action. However, each modification is only 4 operations long, that is, the 4 operations that correspond to the specific chosen link. Thus, the total number of operations is $n * |E| + 4 * n|E| = 5n|E|$.

Object	Number of Entries	Size of Entry	Total size
MDP State	$\|E\| \times 4$	32 bit	$\|E\| * 16$
Trilattices	$\|E\| \times M \times \|E\| \times T_{stable}$	32 bit	$\|E\| \times M \times T_s table * 4$

Table 5.1: *Policy Memory Requirements. For a fully connected network of 10 nodes, there are 45 links. If the total number of bandwidth is 20 levels, then the T_{stable}, that is, the time level where the trilattice no longer changes, is around 1000 time steps. So, the total memory requirements are 45*16 + 45*20*1000*

CHAPTER 6

Numerical Evaluation

All experiments were run on 32 bit Pentium IV 3.0 GHz CPUs with hyper-threading technology equipped 4 GB RAM and 250 GM hard disk. The running environment is the Linux distro Ubuntu 8.06. The programming language used is Matlab R14SP3 interfaced with the Mosek v4 optimization toolkit. The total number of constraints that Mosek can handle in a 32 bit Linux environment is approximately 1,000,000. Even though in the original version in [6] the number of constraints is 2,000,000, we will only use 1 million in all BRLP runs. Fortunately, the number of constraints does not affect the average performance of the derived policy, but it does affect the variance in that policy's performance.

6.1 Tetris Results

Recall that for the Tetris optimization, we implemented the BRLP method to allow the following configurations(see Table 4.2):

(a) The Sampling Interval M. How many moves must pass from the time we've recorded a sample, before we record the next one.
(b) The constraint generation strategy. How many constraints are generated for every sample. We can either generate 1 constraint per sampled state and action (1 per (x, a)), where the sampled action is the action chosen by the current policy; or we can disregard the
(c) The ratio of new samples vs old samples in our sample set. To allow for guarded sampling, the implementation can construct the sample set from a combination of new states, sampled according to the current policy, and states taken from the previous sample. The ratio of new the sample set states vs the states taken from the previous sample may be adjusted. In any case, the size of the sample is always sufficient to create 1000000 constraints, that Mosek can handle.

Five methods were tried in total:

(a) Standard BRLP
(b) Guarded Sampling BRLP
(c) Adaptive Sampling BRLP
(d) Conditional Sampling BRLP
(e) Least Square

6.1.1 Standard BRLP

The standard BRLP method is the one that was successfully used in [6]. The method is configured as follows:

(a) Sampling interval M: 90 moves. This is a fixed interval
(b) Number of constraints a sampled state creates : One constraint for every possible action, for every sampled state. There are on average 23 possible combinations of rotations and translations that we can perform on a falling piece, thus , for every sampled state there are on average 23 actions that can happen on that stage (see Table 4.1.2.
(c) Number of states sampled in each BRLP iteration(New:Old) : Every iteration generates an entirely new sample of states. The sample must be large enough to generate 1000000 constraints. For every sampled state, we generate 22 constraints on average, meaning that we require approximately $\frac{1000000}{23} \approx 45500$ sampled states.

The BRLP method was repeated a total of 6 times, using 6 different starting policies. Each run of the BRLP lasted 5 iterations. Each iteration was repeated 5 times, in order to create 5 different samples. From the 5 samples, we generated 5 different policies and the policies were evaluated by taking the average score over 100 games. The best policy was used in the next iteration. Thus, a total of $6 * 5 * 5 = 150$ samples were generated, each sample holding 1 million constraints. Table 6.1 shows some statistics from the system run.

Total # of Sample sets	Average Sample size (in States)	Total # of moves performed	Total # games played
150	46 983	634 270 500	279 718

Table 6.1: *BRLP Statistics*

Unfortunately, we were unable to generate the results reported in [6] using this configuration. This is a testament that the BRLP method's success depends on the initial policy. The graph 6.1 shows the most successful run of a BRLP iteration.

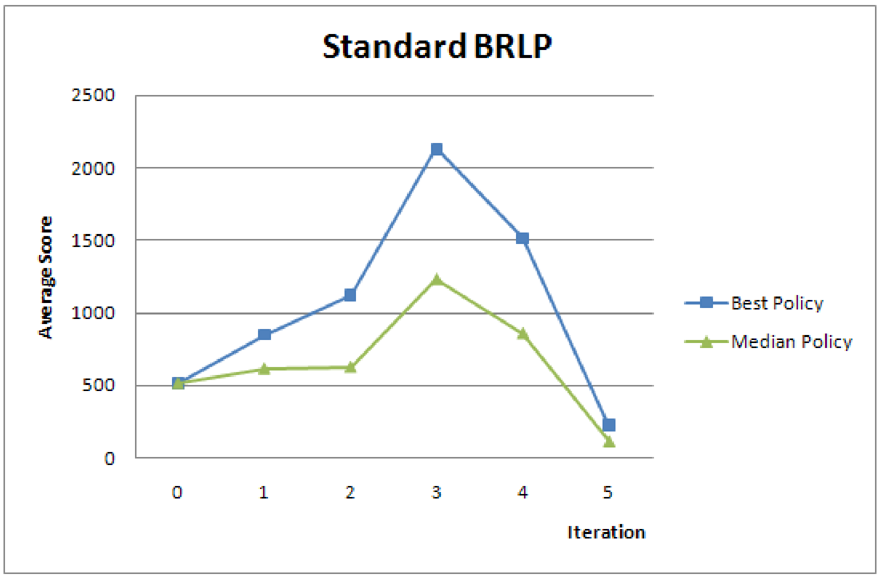

Figure 6.1: *The most successful run for the standard BRLP. The graph shows the average score achieved by the best policy derived in each iteration. It also shows the score of the median policy. The best policy achieves scores well below the scores that were reported in [6]*

Guarded Sampling BRLP

In the Guarded Sampling version of the BRLP, we create a sample using states that are sampled by the current policy, and states that exist in the previous sample. For this method, we tried different proportions of new states vs previous states. The method's exact configuration is as follows:

(a) Sampling interval M : 90 moves. This is a fixed interval

(b) Number of constraints a sampled state creates: For this method, we have tried both configurations (1 constraint for every possible action per state, and 1 constraint per state-action pair in the sample). In the case where we generate one constraint per state, that constraint corresponds to the action that is chosen by the policy. This way, we can use more states in our sample. However, generating only 1 constraint per sample, increases the number of states that we need to sample by a factor of approximately 23. This has a major impact on the time that it takes an iteration to complete.

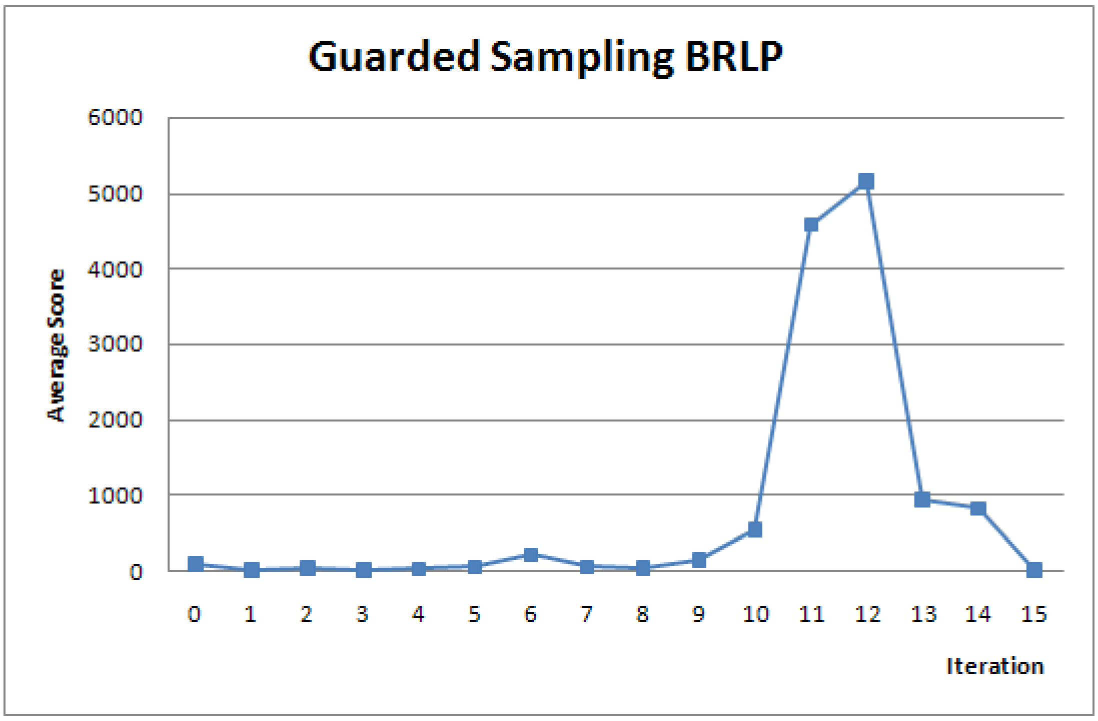

Figure 6.2: *The successful application of the guarded BRLP method. The graph shows the average score of the different policies at each iterations. The method used for this graph was configured for 1 constraint for every possible action per sampled state, and a sampling mix of 250 000 new samples and 750 000 old samples.*

(c) Ratio of new states against old states in the sampling mix: We have tried 3 different combinations:

- 250 000 constraints generated new samples, the remaining 750 000 are taken from the previous sample
- 500 000 new constraints vs 500 000 from the previous sample
- 750 000 new constraints vs 250 000 from the previous sample

In this case, we did not repeat each iteration in order to choose the best policy to continue. Instead, we left the BRLP method to run for up to 20 iterations. This method proved a success, as we managed to generate a policy with a score of approximately 6000 lines per game, on average. The successful sample mix is the one where 250 000 new constraints are generated by the current policy, and 750 000 are taken from the previous sample. With this mix , we were able to derive various hi-scoring policies, with performance ranging from 4200 to the maximum of 6000 lines, using different starting policies. Both configurations of 1 constraint per sampled state-action pair and one constraint per sampled state for all possible actions succeeded.

These results are very encouraging for the Routing problem, because they provide evidence that :

(a) the BRLP method can give a good policy if its let to run long enough, regardless of the initial policy
(b) there is no need to generate one constraint per possible action for each sample state. The method also works when we generate a single constraint for each sampled state, that corresponds to the action chosen by the policy in use.

6.1.2 Adaptive Sampling BRLP

Once we'have found the optimal policy, we investigate why the performance reaches a peak during the BRLP iterations. One possibility is that samples that are 90 moves apart become correlated as the policy gets better. A similar possibility is that under a better policy, the needed samples are generated in a much smaller number of games, and thus there are less opportunities for a bad or terminating state to be sampled. This is illustrated in Figure 6.3.

The adaptive sampling BRLP is configured so that the number of games needed to generate the sample remains the same through all BRLP iterations. Apart from the sampling interval, all other parameters including the starting

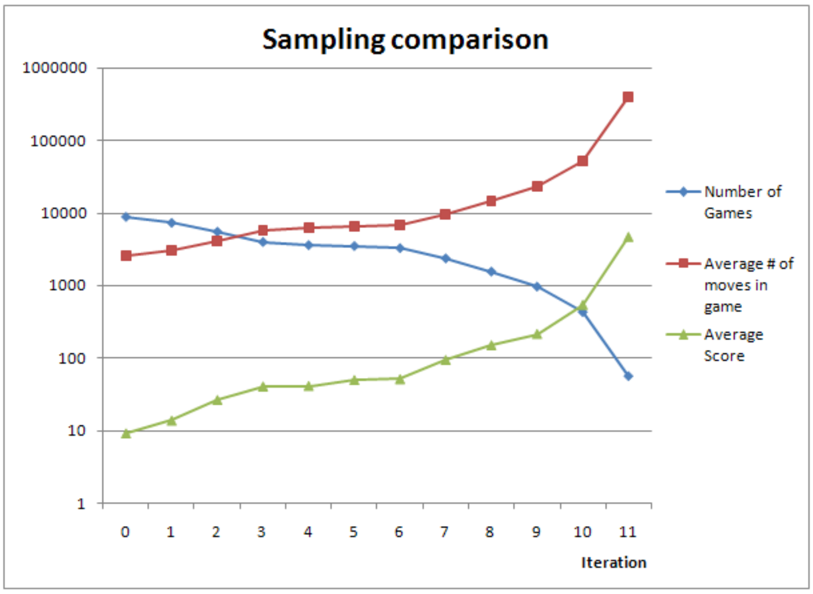

Figure 6.3: *The graph shows the number of games needed to collect 250 000 samples, as it was recorded during a BRLP run, where the sampling interval was fixed at 90 moves. The y axis is in logarithmic scale.*

policy, are the same as the ones that yielded the successful policy . The adaptive sampling version of the BRLP method is configured as follows:

(a) Sampling interval M: Varies according to the policy. At each iteration, there is a calibration stage where the policy is used for 100 games, without any sampling occurring. During that stage, the total number of moves played through the 100 games is recorded and the average number of moves per game is derived. The goal is to play 2000 games every iteration, so the expected number of moves that will happen when that policy plays 2000 games is estimated. The expected number is divided by the 250 000 samples that we need, and thus we get the sampling interval. The sampling interval is not allowed to go below 90, and so, for very bad policies we need to play more than 2000 games.

(b) Number of constraints a sampled state creates: We generate 1 constraint for every possible action for each sampled state. Thus, we generate approximately 23 constraints per sample.

(c) Number of new constraints generated in each BRLP iteration : 250 000 new constraints generated by current policy, the remaining 750 000 are taken from the previous sample

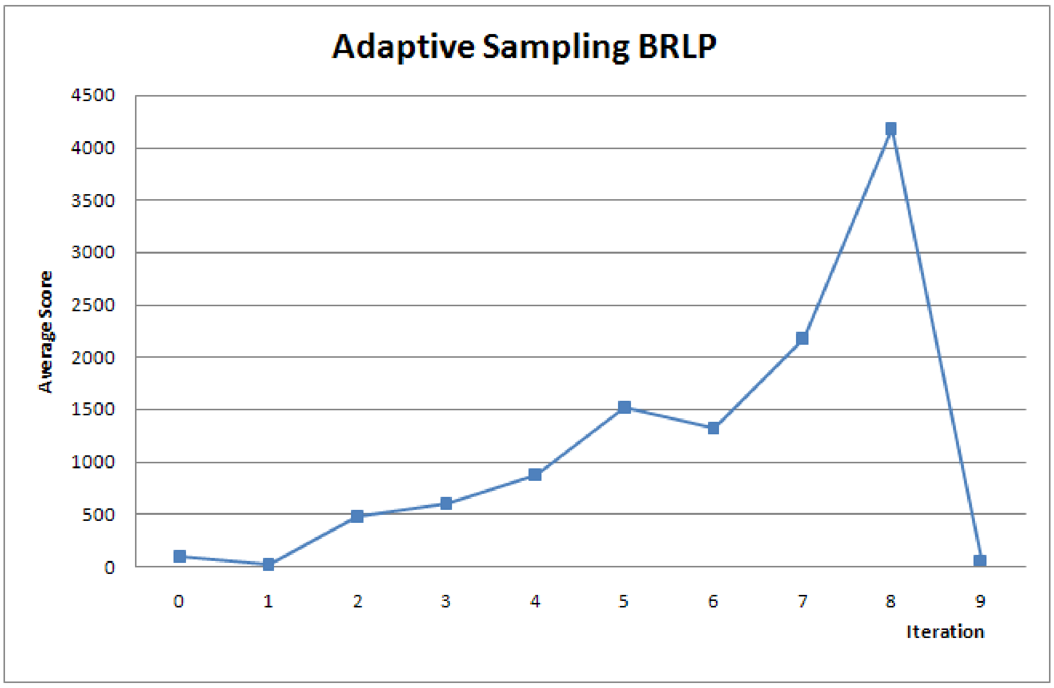

Figure 6.4: *A successful run of the adaptive sampling BRLP, showing the average score achieved by the policies at each iteration.*

The adaptive sampling technique behaved the same way as the other techniques. The performance of the policies generated reached a peak at roughly the same level as before. However, there was a major impact in the performance of the BRLP method. As the sampling interval increases, the BRLP method requires a lot more time to generate the samples. This has a major impact on the applicability of the method, especially in larger problems where more samples will be needed. Table 6.2 shows how the sampling interval was set in each of the iterations.

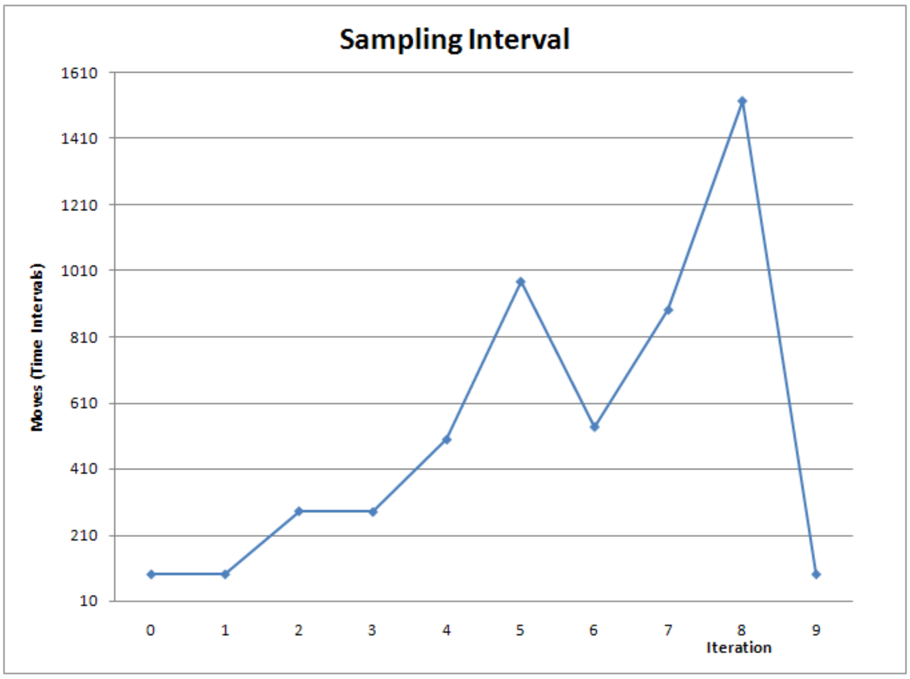

Figure 6.5: *The graph shows how the sampling interval is set in each iteration. The y axis represents moves, which are equivalent to time intervals in the MDP process. Table 6.2 shows the data in more detail.*

Iteration	Avg Moves per Game	Proposed Sampl. Interval	Actual Samp. Interval	Avg Samples per game point
0	5064.12	40.51	90	53.73
1	2647.38	21.18	90	147.08
2	35186.33	281.49	281	74.08
3	34869.36	278.95	279	57.54
4	62315.05	498.52	499	70.89
5	122161.96	977.30	977	80.21
6	67113.99	536.91	537	50.96
7	111448.17	891.59	892	51.31
8	190660.04	1525.28	1525	45.59
9	3425.86	27.41	90	68.52

Table 6.2: *The sampling interval is adjusted, so that at each iteration a minimum of 2000 games is required to generate 250 000 constraints to use in the new sample.*

6.1.3 Conditional Sampling

For conditional sampling, we need to input the following:

(a) The optimal policy found so far, to be used as initial policy u_k
(b) The sample that derived that optimal policy, to be used as the base sample

The conditional sampling method, does not replace any of the existing samples. Instead it only adds the generated samples. As soon as there is a new sample, the simulation ends and the optimization is started to derive the new policy. A new sample is generated when, during the simulation, we encounter a state that under the present policy violates the constraints. That is, assume that the present policy u_k operates according to the weight vector r_k, and for the state x chooses an action a. Then, the state-action pair (x, a) will be entered in the sample if it violates the constraint:

$$g_a(x) + \alpha \sum_{y \in S} P_a(x, y)(\Phi r_k)(y) \geq (\Phi r_k)(x) \tag{6.1.1}$$

Adding just 1 constraint to the sample changes the solution dramatically. That is, from the first iteration where the sample used is the one that results to the optimal policy, we add to that sample the first state that is violated by the optimal policy and we solve the RLP. The derived policy is dramatically worse than the policy that would have been derived without that constraint (i.e. the optimal policy). Figure 6.6 demonstrates the score of the two policies over 100 games.

This is a surprising result, as it seems to suggest that the BRLP method benefits from the fact that it does not include all the constraints in the problem. Such a situation is illustrated in figure 6.1.3. The figure shows an MDP process with two states, J_1 and J_2 which correspond to the axes. The original problem with all the constraints is shown on the left hand side. The feasible region is represented as the gray area. The exact solution to the problem, and thus the true-cost-to-go approximation is J^*. However, the solution of the approximate linear program lies on the approximation space shown as the line Φr. Given that choice of approximation, we see that the best approximation lies at the point which is closest to J^*, namely point Φr^*. However, when using the LP approach, the approximation we get lies at the maximum intersection of the approximation space (line Φr) and the feasible area. That intersection is labeled $\Phi \widetilde{r}$. It is clear that the constraint labeled A renders the constraint B (shown as the doted line)inactive. That is, as long as A is included constraint B will not influence the solution. However,consider the reduced LP on the right hand side. Constraint

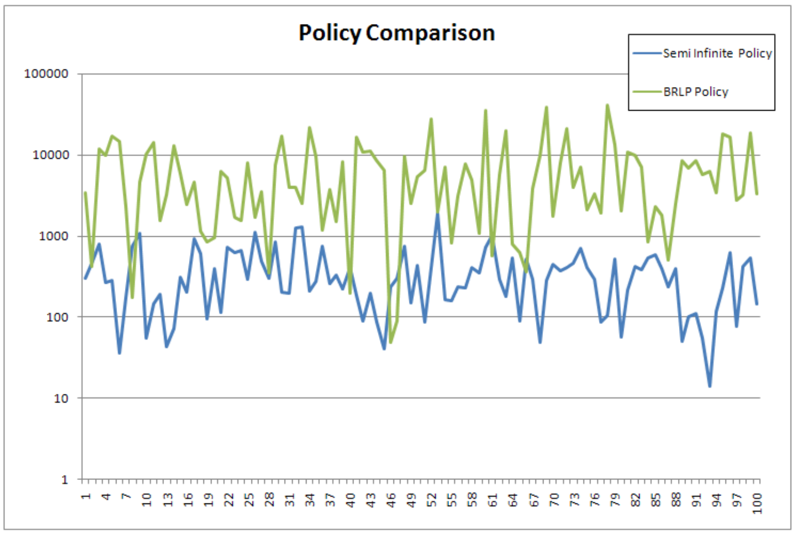

Figure 6.6: *The performance difference of the Semi Infinite Policy and the BRLP policy. The samples that generated the two policies vary only in a single constraint*

A is not included in the problem. That is, suppose at an iteration of the BRLP, we do not sample constraint A, but we sample constraint B. The feasible region is again shown in the shaded area, however, the solution of the reduced Linear Problem will give a point a lot closer to the real cost-to-go, than before. Thus, we have actually benefited from the fact that we did not include constraint A.

This observation suggests a change in the sampling mechanism. Instead of attempting to find a sample that will approximate the entire LP, we should aim for a sample that will force the LP to come as close to the true cost-to-go as possible. However, more rigorous analysis and investigation is required.

6.1.4 Least Square Objective

In order to test whether the above findings are justified, we decided to reformulate the problem as a least square problem. That is, instead of trying to solve the RLP at each iteration, we will try to find the weights that make the RHS in Bellman's equation equal to the LHS(see subsection 2.4.2). Recall that Bellman's equation for a parametric approximation of the cost-to-go is:

$$(\Phi r)x = \min_{a \in A_x} \left\{ g_a(x) + \alpha \sum_{y \in S} P_a(x,y)(\Phi r)(y) \right\} \tag{6.1.2}$$

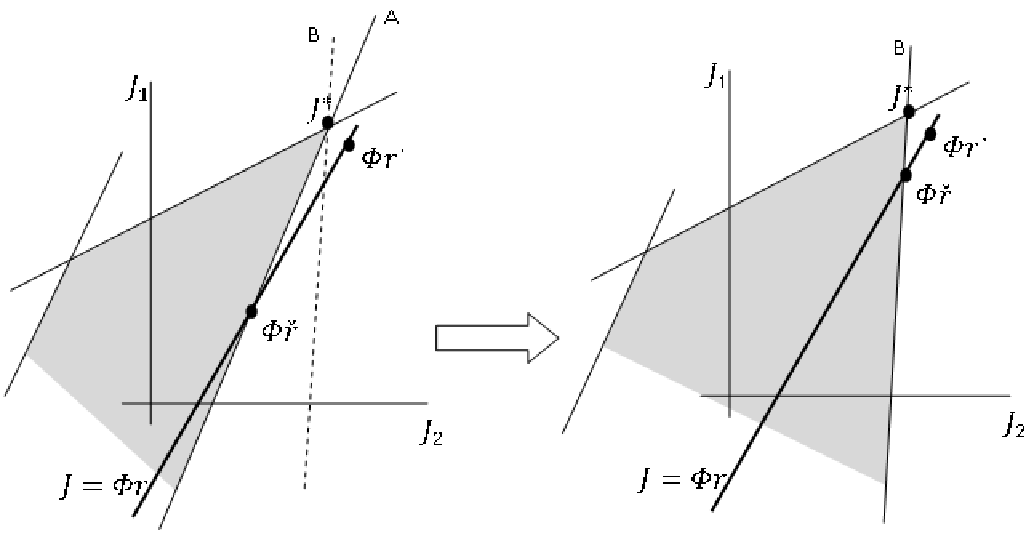

Figure 6.7: *Graphical illustration of the solution of an RLP with missing constraints (on the right), compared to the solution of the actual LP which includes all the constraints.*

Using the l_2-norm, we want to minimize the LHS and the RHS for the state and action in our sample set:

$$\min_r \sum_{(x,a)\in\bar{X}} \left((\Phi r)(x) - \left(g_a(x) + \alpha \sum_{y\in S} P_a(x,y)(\Phi r)(y)\right)\right)^2 \tag{6.1.3}$$

By differentiating the above, we can derive the r vector that makes the derivative 0, and thus minimizes the (6.1.3). The vector is found to be:

$$\begin{aligned} r = & \sum_{(x,a)\in\bar{X}} (d_{a,x}\, d_{a,x}^T)^{-1} \sum_{(x,a)\in\bar{X}} (g_a(x)\, d_{a,x}) \\ \textit{where} & \\ d_{a,x} = & \ \Phi(x) - \alpha \sum_{y\in S} P_a(x,y)\Phi(y) \end{aligned} \tag{6.1.4}$$

The BRLP method remains the same, except that now instead of solving the RLP, we solve the equation (6.1.4). The method is summarized as follows:

1. Begin with a simulator that uses a policy u_0
2. Generate a sample $\overline{X_k}$ using policy u_k
3. Solve the Least Square problem (6.1.4) according to the sample $\overline{X_k}$ to generate a policy u_{k+1}

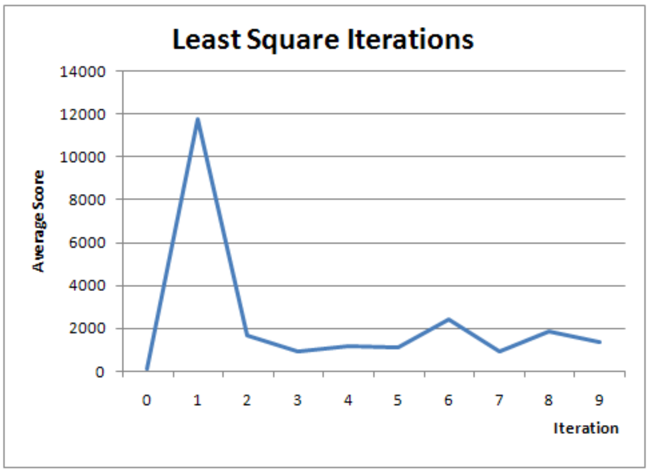

Figure 6.8: *The average performance of the policies at each iteration*

4. Increment k and go to step 2

The results are shown in graph 6.8. As mentioned in 2.4.2, the least square methods are usually one step methods, that is, they succeed on the first iteration. This is evident in the graph.

The policy derived from the Least Square method scores on average 11000 lines. The policy is almost two times better than the policy which resulted from the BRLP method. Figure 6.9 shows the performance of the best BRLP policy and the Least square policy over the course of 100 games.

Graphs 6.10 and 6.11 show the different approach that the BRLP method and the Least Square method take in order to solve Bellman's equation. The graphs shows the right hand side and left hand side of the constraints evaluated for the two policies over a sample set of states and actions. Observe that in the BRLP version, the RHS is always smaller from the LHS is negative, as depicted by the constraints in the RLP. Thus, we can visualize the BRLP method as trying to approach the solution from below. On the other hand, the Least Square version the difference oscillates around the x-axis. This is because the Least Square method tries to minimize the distance between the two, regardless of whether the LHS is larger or smaller from the RHS.

- RHS : $(\Phi r_{pol})x$
- LHS : $g_a(x) + \alpha \sum_{y \in S} P_a(x, y)(\Phi r_k)(y)$

where (x, a) is the sample.

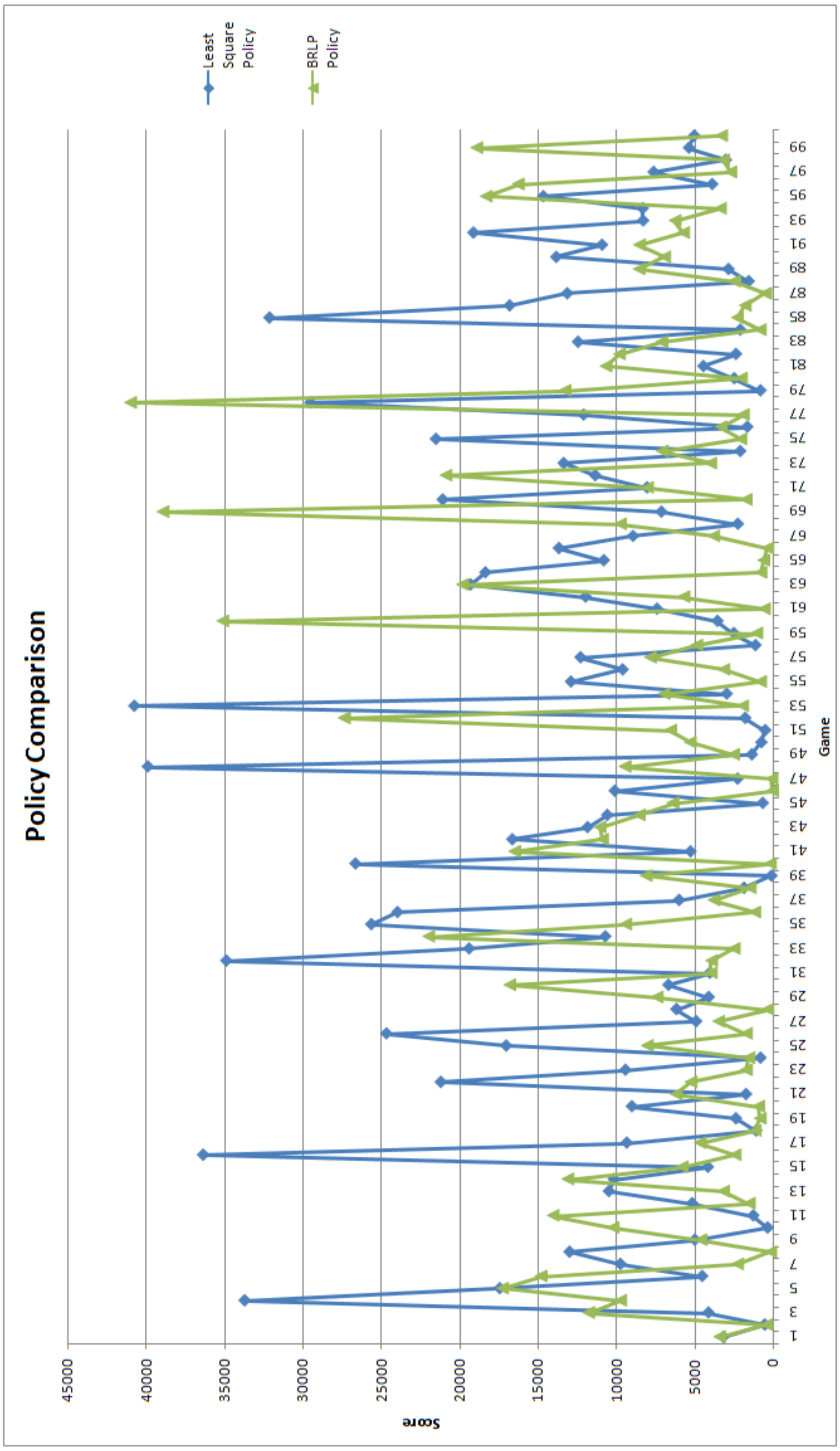

Figure 6.9: *The two optimal policies compared over 100 games*

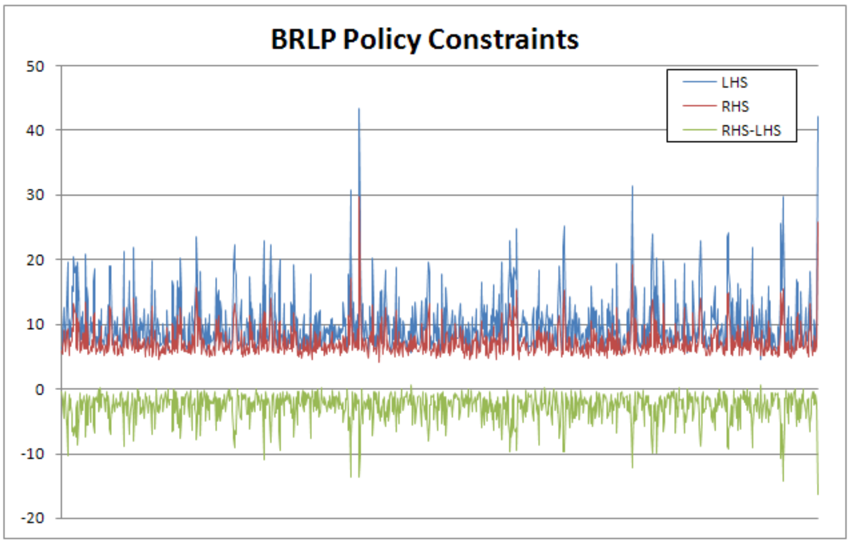

Figure 6.10: *The constraint samples of the BRLP policy. The RHS corresponds to* $(\Phi r_{BRLP})x$ *and the LHS to* $g_a(x) + \alpha \sum_{y \in S} P_a(x, y)(\Phi r_k)(y)$ *,where* (x, a) *is the sample. The RHS is always smaller that the LHS, as this is what is required by the RLP being solved.*

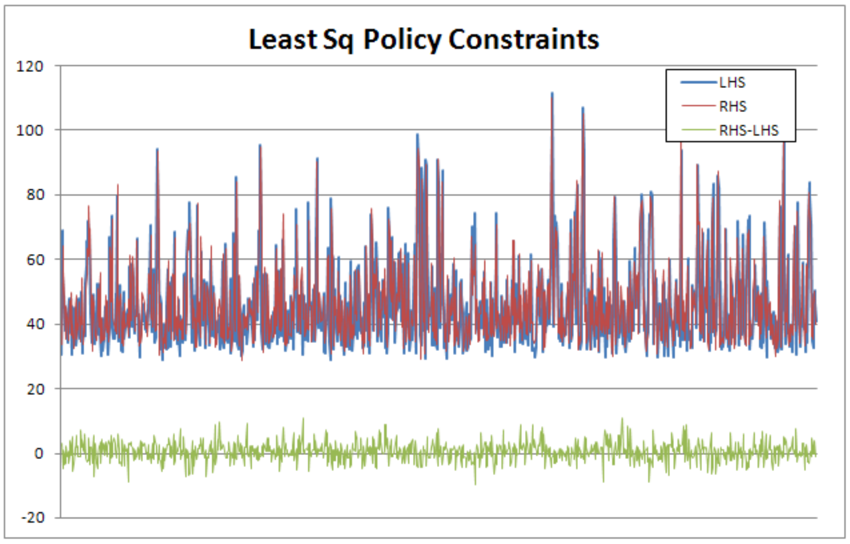

Figure 6.11: *The constraint samples of the Least Square policy. The RHS corresponds to* $(\Phi r_{LSq})x$ *and the LHS to* $g_a(x) + \alpha \sum_{y \in S} P_a(x, y)(\Phi r_k)(y)$ *,where* (x, a) *is the sample.*

6.2 Routing Results

All experiments were done on fully connected networks, of a total of 10 nodes. The reason we have chosen to do the experimentation on fully connected networks is that they represent overlay networks, which this kind of system might be applied. However, each network has different distributions for each of its link, which are generated randomly using a Matlab program.

We attempted to optimize the routing problem using 2 methods:

(a) Guarded Sampling BRLP
(b) Least Square

The two methods yielded almost identical results, so we will only present the best results for each network. The sampling configuration that worked best for the BRLP method, and was that used in both methods is as follows:

(a) Sampling Interval M: 120 fixed
(b) Constraints Generated per sample: 1 constraint for each sampled state-action pair
(c) Number of new constraints generated in each iteration : 250 000 new constraints generated by current policy, the remaining 750 000 are taken from the previous sample

The maximum bandwidth levels were set 20.

The graphs below show the results. We were unable to generate a heuristic policy that would produce good results, due to the complexity of the system. There are 45 links in a fully connected network requiring $4*45+1 = 181$ weights meaning that handcrafting a weight vector for those weights is not trivial. Also,as this is a fully connected, the links have the same structural properties,i.e. they are part of the same number of paths. The only difference between the links is their bandwidth distribution, making the task of handcrafting the vector even harder. As a result, the performance comparison was done against the round robin policy and the perfect information policy.

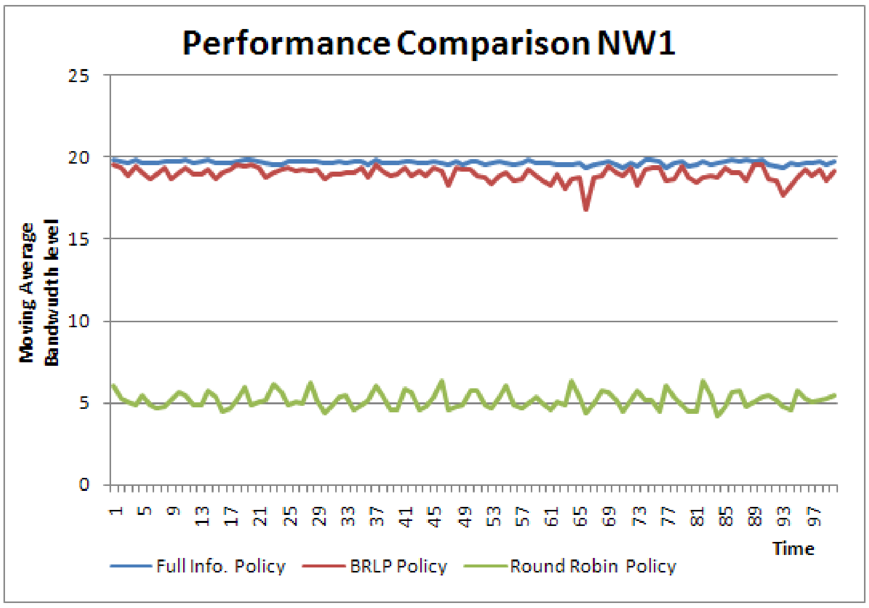

Figure 6.12: *Performance for Network Configuration 1*

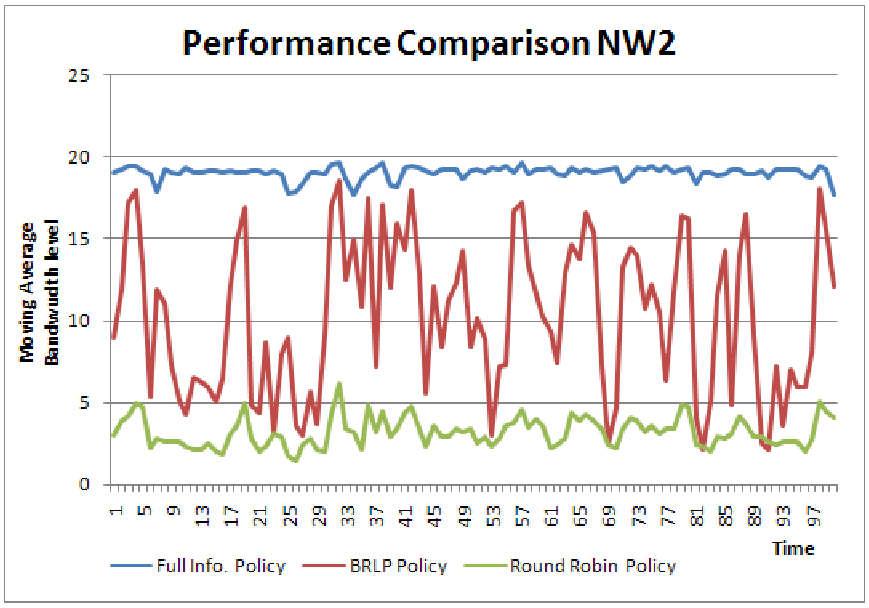

Figure 6.13: *Performance for Network Configuration 2*

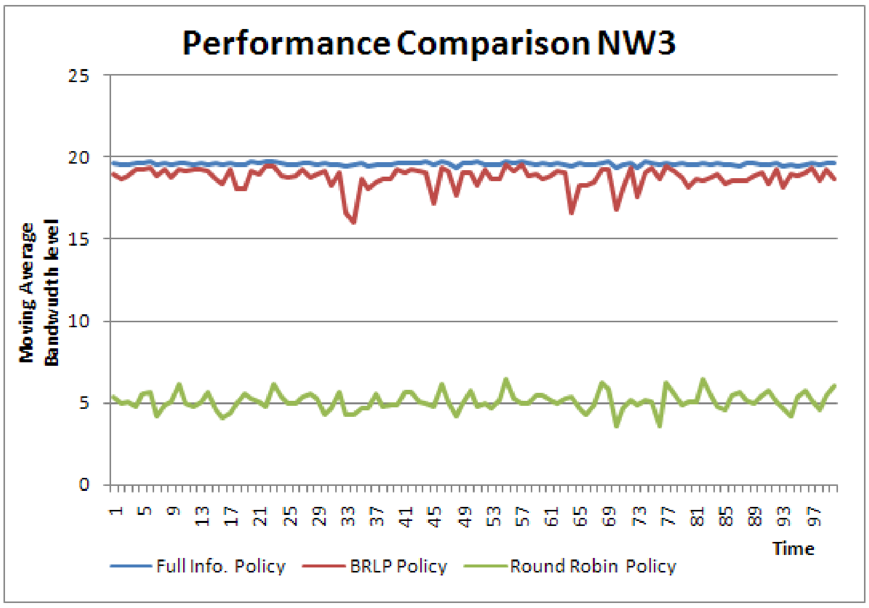

Figure 6.14: *Performance for Network Configuration 3*

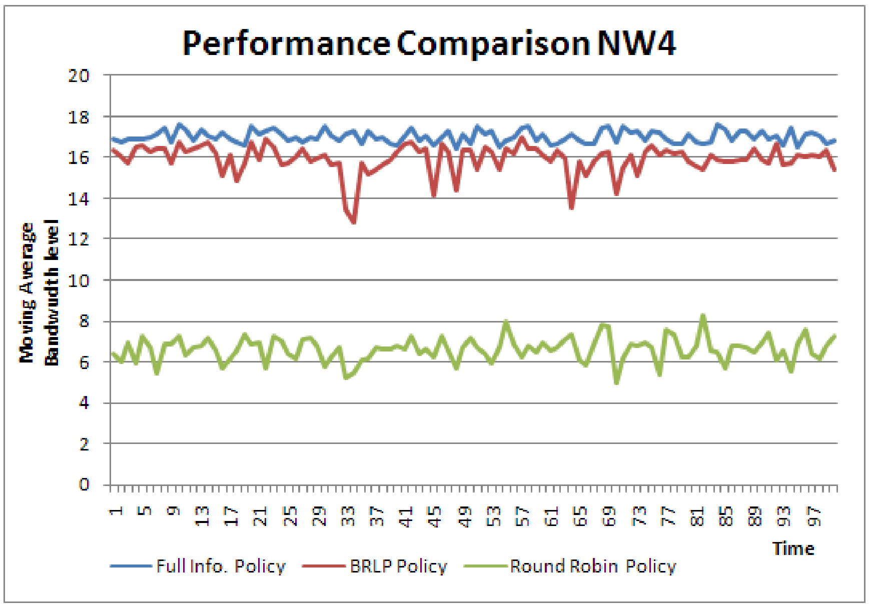

Figure 6.15: *Performance for Network Configuration 4*

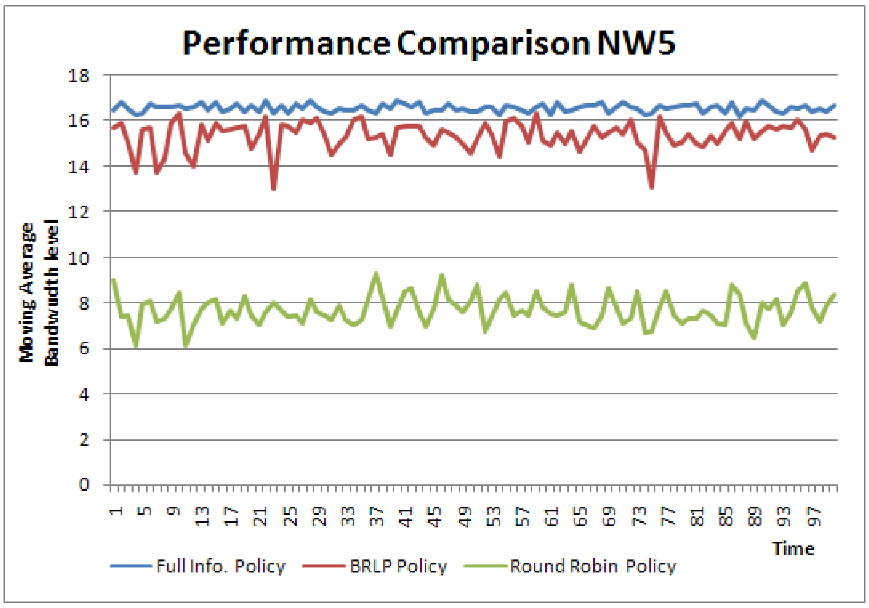

Figure 6.16: *Performance for Network Configuration 5*

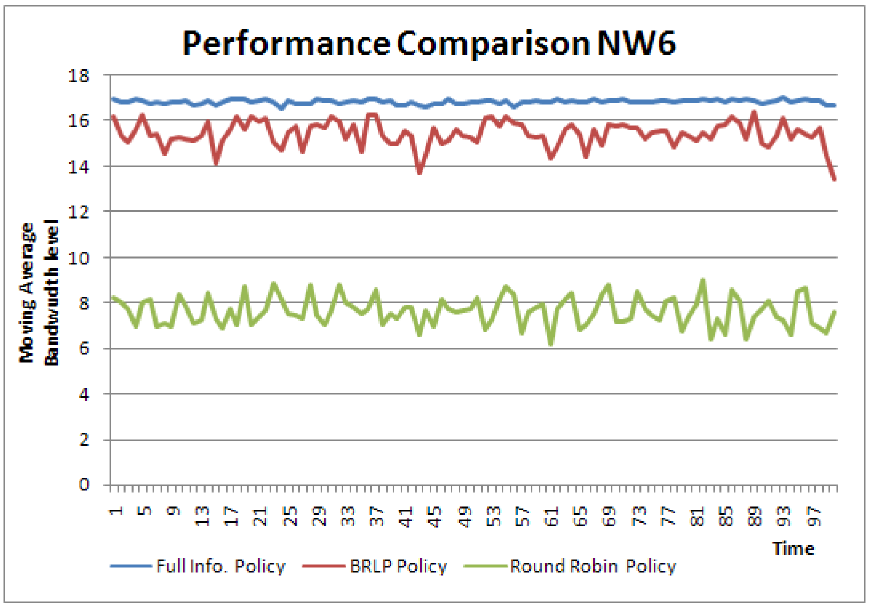

Figure 6.17: *Performance comparison for Network Configuration 6*

CHAPTER 7

Conclusion and Future Work

The BRLP method is a very powerful approximate dynamic programming technique. It is designed to overcome the curse of dimensionality and can be used to optimize stochastic problems which cannot be solved using exact dynamic programming. However, the success of the method is highly dependent on the running parameters, such as the starting policy, the sampling interval and the mix of old and new constraints in the sample sets.

By fine-tuning the method, we have managed to generate a policy for Tetris that is better than what was originally though possible. That is, we have managed to bumped the performance of the optimal policy derived by the BRLP, from 4300 points per game to 6000 points. Additionally, we have provided a possible reason as to why the BRLP method seizes to refine the solution after a certain number of iterations. Namely, we have investigated the possibility of the BRLP method benefiting from the fact that some of the constraints are not included in the RLP problem that is being solved. That is, by excluding certain constraints the BRLP method is able to approach the true cost-to-go a lot more closer, than when we include the constraints. This might explain the reason why there is a performance drop in the policies generated by the BRLP after a small number of iterations. The above finding has inspired as to use a Least Square method to optimize the Tetris problem. The method resulted in a policy that scores an average of 11000 lines per game.

Finally, we were able to formulate the problem of routing under uncertainty as a MDP problem and solve it using the BRLP. The aim was to design a policy that would choose the link to measure before transmission, so that we maximize the throughput of the network using those measurements. The BRLP method was used to optimize a fully connected network of 10 nodes with very successful results. For most network configurations, the policy derived by the BRLP achieved a throughput that was very close to the maximum possible throughput given a perfect set of information. The BRLP policy outperformed a naive Round Robin policy by a significant factor. The results suggest that the BRLP method can be used to optimize overlay networks with similar characteristics.

7.1 Future Work

As future work, we suggest continuing the investigation as to why the BRLP's performance peaks after a certain number of iterations. The investigation can continue with more experimental work, but it should also extend to providing mathematical insights as to why this happens.

For the routing problem, we have shown that the BRLP method is successful for a certain network topology. Future work should broaden this investigation to various other network topologies, and use additional optimization techniques. The method can also be tested in situations where all paths can be used concurrently for transmission, instead of using just the one. Other extensions may include optimizing a network were we have no prior knowledge for the links' bandwidth distribution. The ultimate objective is to implement such a controller on a real overlay network, in order to regulate its traffic, improve its throughput and make it more robust.

Bibliography

[1] Arnab Nilim, Laurent El, and Ghaoui Vu Duong. Management systems (atms) in us and europe.

[2] Evdokia Nikolova and David R. Karger. Route planning under uncertainty: The canadian traveller problem. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 969–974. AAAI Press, 2008.

[3] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Two Volume Set.* Athena Scientific, 2007.

[4] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming (Optimization and Neural Computation Series, 3).* Athena Scientific, May 1996.

[5] I. Szita and A.Lorincz. Learning tetris using the noisy cross-entropy method. *Neural Comput.*, 18(12):2936–2941, 2006.

[6] Vivek F. Farias and B Van Roy. Tetris:a study of randomized constraint sampling. 2006.

[7] Vivek F. Farias and Benjamin Van Roy. Tetris: Experiments with the lp approach to approximate dp, 2004.

[8] Bellman R E. Dynamic programming. Technical report, Princeton University Press, 1957.

[9] Bellman R E. A marcovian decision process. *Journal of Mathematics and Mechanics*, 1957.

[10] Warren B. Powell. Approximate dynamic programming: lessons from the field. In *WSC '08: Proceedings of the 40th Conference on Winter Simulation*, pages 205–214. Winter Simulation Conference, 2008.

[11] Michael C. Fu. Simulation optimization. In *WSC '01: Proceedings of the 33nd conference on Winter simulation*, pages 53–61, Washington, DC, USA, 2001. IEEE Computer Society.

[12] D. P. de Farias and B. Van Roy. The linear programming approach to approximate dynamic programming. *Oper. Res.*, 51(6):850–865, 2003.

[13] D. P. De Farias and B. Van Roy. On constraint sampling for the linear programming approach to approximate dynamic programming. *Mathematics of Operations Research*, 29:2004, 2001.

[14] R. Hettich and K. O. Kortanek. Semi-infinite programming: theory, methods, and applications. *SIAM Rev.*, 35(3):380–429, 1993.