On the Complexity of the Policy Improvement Algorithm for Markov Decision Processes

Mary Melekopoglou
Anne Condon[1]

Computer Sciences Department
University of Wisconsin - Madison
1210 West Dayton Street
Madison, WI 53706

ABSTRACT

We consider the complexity of the policy improvement algorithm for Markov decision processes. We show that four variants of the algorithm require exponential time in the worst case.

1. Introduction

Finding an optimal policy in a Markov decision process is a classical problem in optimization theory. Although the problem is solvable in polynomial time using linear programming (Howard [4], Khachian [7]), in practice, the policy improvement algorithm is often used. We show that four natural variants of this algorithm require exponential time in the worst case.

A stationary Markov decision process consists of a finite number of states, which includes an initial state s_0. At each time t=0,1, $\cdots$, the process is in some state s_t. For each state s there is a finite set of decisions D_s. If the process is in state s at time t and a decision d in D_s is chosen, there is a fixed transition probability $p(s,s',d)$ of being in a state s' at time t+1. The decision d incurs a cost $c(s,d)$. A policy S is a mapping from states to decisions. The cost of a policy can be measured in different ways (see Howard [4]). The discounted cost with discount factor b is the expectation of the sum over all t of $c(s_t, S(s_t))b^t$. The total cost is the expectation of the sum over all t of $c(s_t, S(s_t))$. The average cost is the limit as t goes to infinity, of the total cost up to time t divided by t+1. The discounted cost is always finite, whereas the other measures may be infinite. For a given cost measure, a policy is optimal if it minimizes the expected cost.

The policy iteration algorithm is an iterative procedure that finds an optimal policy within a finite number of steps, which is at most exponential in the number of states. Assume for the moment that the cost

1 Supported by National Science Foundation grant number CCR-8802736.

measure is the discounted measure, since this is always finite. Roughly, the algorithm proceeds as follows. With respect to a given policy, we say that a state is switchable if the policy obtained by changing the decision at that state improves the policy, that is, lowers its cost. Initially, choose an arbitrary policy to be the current policy. Improve the current policy by switching the decision at one or more switchable states. Repeat this until no state is switchable. At that point, an optimal policy is reached. Since the number of policies is at most exponential in the number of states, the algorithm halts within exponential time. See Howard [4] for a proof of correctness of this algorithm. We describe this algorithm in more detail in the next section.

There are many versions of this algorithm, depending on how the states to be switched are selected at a step of the algorithm. In this paper, we consider four versions and prove that they require exponential time in the worst case. To do this, we present for each algorithm an example of a Markov decision process on which the algorithm takes exponential time.

2. Exponential Time Examples for the Policy Improvement Algorithm

Our examples are represented as graphs, where vertices represent states and edges represent transitions. The graphs have two types of vertices, called min and average vertices. Min vertices have two decisions, 0 and 1, and on each there is one transition with probability 1. Average vertices have one decision, with two transitions, each with probability 1/2. There are two special sink vertices, each with no outgoing edges. All edges have cost 0, except edges to the 1 sink, which have cost 1. Thus the total cost of a policy is the probability of reaching the 1-sink vertex.

The **cost** of a vertex *i* with respect to a policy *P* is the probability of reaching the 1-sink from i, when the policy is *P*. (Note that this cost is the total cost of the process, as defined in the previous section). A min vertex is **switchable** with respect to a fixed policy if its cost is not the minimum of the costs of its children.

On these graphs, the policy improvement algorithm works as follows. The algorithm starts from an initial policy. Then it repeatedly selects a switchable min vertex, and **switches** it. That is, it changes the policy so that the edge corresponding to the selected vertex is replaced by the other edge from the vertex. The algorithm halts when there is no switchable vertex, in which case an optimal policy has been found. For our graphs, the policy is optimal with respect to any of the three cost measures defined in the previous section. More than one vertex may be switchable at any iteration. Thus, to completely specify the algorithm, a **select procedure** must be defined, which returns the next vertex to be switched, if any. We now describe four select procedures and later we present exponential time examples for each of these procedures.

The select procedure of the **Simple Policy Improvement Algorithm** selects arbitrarily one of the switchable min vertices. Specifically, it selects the switchable vertex with the largest number. The algorithm is simple in the sense that the select procedure does not select the vertex to be switched based on

either the costs of the vertices or the structure of the graph.

The select procedure of the **Topological Policy Improvement Algorithm** needs some preprocessing of the graph. The vertices are topologically sorted, that is, an integer order is assigned to each min vertex, such that if there is a path from vertex *i* to vertex *j*, then the order of *i* is greater or equal to the order of *j*. The graph is in this way divided into components of vertices with the same order. The select procedure selects the largest numbered switchable vertex in the component of lowest order that contains switchable vertices. In this way, when a vertex *i* is switched, there is a decrease in the cost of every vertex with larger order from which there is a path to *i*, according to the current policy. This algorithm can perform a lot better than the simple algorithm. If a directed graph is divided into *k* strongly connected components, and the numbers of min vertices in them are $n_1, n_2, ..., n_k$, then, in the worst case the algorithm takes $2^{n_1} + 2^{n_2} + \cdots + 2^{n_k}$ steps. If the maximum size of a component is bounded by a constant, the algorithm can find a best policy in a polynomial number of steps. In particular, if the graph is acyclic the algorithm makes at most linear number of switches.

We now describe the select procedure of the **Difference Policy Improvement Algorithm.** As in the topological policy algorithm, the vertices are first topologically ordered. Consider the min vertices in the component with the lowest order that contains switchable vertices. For these vertices, the select procedure computes the difference between the costs of its two children, and selects the vertex with the largest difference. (If more than one vertex has this difference, the largest numbered one is selected). This is a natural approach to the problem of reducing the maximum number of switches required by the policy improvement algorithm, because when a vertex with a large difference is switched, the decreases of the costs of the vertices with larger order may be large too.

Finally, we describe the **Best Decrease Policy Improvement Algorithm.** The select procedure of this algorithm first topologically sorts the vertices of the graph. Again, consider the switchable min vertices in the component of lowest order that contains switchable vertices. For these vertices, the select procedure computes the decrease of the cost of the vertex, if it is switched. The vertex that will get the largest decrease is selected (if more than one vertex can get this decrease, the largest numbered one is selected).

We now present exponential time examples for these four algorithms. We use the following conventions in presenting our examples. A policy for a graph with *n* min vertices is represented as an *n*-bit vector, $S = S_n S_{n-1} \cdots S_1$, where S_i is the label of the edge in *S* that originates from *i*. For simplicity, when describing the execution of a policy improvement algorithm, we denote by $V(i)$ the cost of vertex *i* with respect to the current policy. In our figures, min vertices are represented with circles numbered 1, 2 , ..., average vertices with circles numbered 0′, 1′, ..., and the sink vertices with squares. The two edges for every min vertex are labeled with the decisions 0 and 1.

2.1. The Simple, Topological and Difference Algorithms

We first consider the difference policy improvement algorithm. Our exponential time example for this algorithm is also an exponential time example for the simple and topological policy improvement algorithms. We first present a basic graph and then we add a gadget in appropriate places of this graph. The basic graph, given in Figure 2.1, has n min vertices (labeled 1, 2, ..., n), $n+1$ average vertices ($0'$, $1'$, ..., n'), and the two sinks.

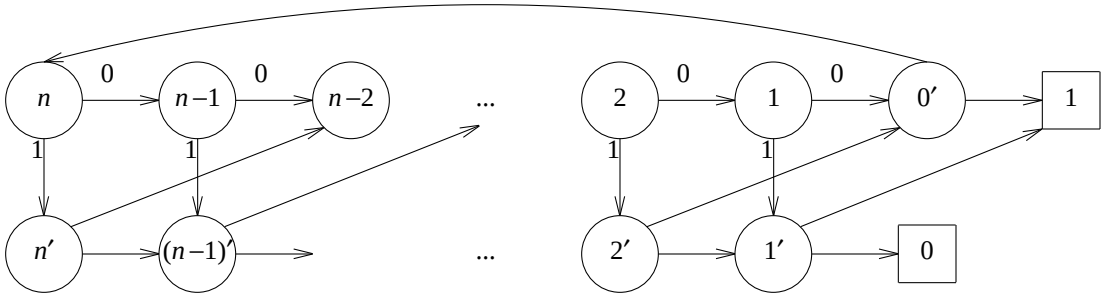

Figure 2.1: Basic graph for general n.

The structure can be described recursively; to construct the basic graph for $n+1$, given the graph for n, we add a new min vertex, vertex $n+1$, and a new average vertex, vertex $(n+1)'$. The two edges for vertex $n+1$ lead to vertex $(n+1)'$ and vertex n. Vertex $(n+1)'$ is the average of n' and $n-1$ (or $0'$ if n is 1). Also, there is an edge from $0'$ to min vertex $n+1$ instead of n.

This basic graph of Figure 2.1 is augmented by adding the **gadget $\mathbf{g_k}$** of Figure 2.2 in appropriate places.

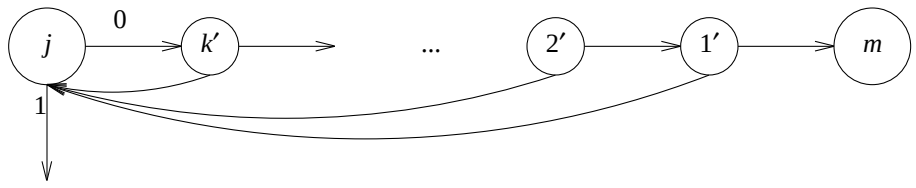

Figure 2.2: Gadget g_k

The gadget g_k is placed between a min vertex j and another, m, which can be a min, average or sink vertex. It consists of k average vertices, $1'$, $2'$, ..., k'. Vertex i' ($1 \le i \le k$) has one edge going to min vertex j and another going to the previous average vertex, $(i-1)'$, or to vertex m if i is 1.

It is easy to notice that this gadget has the following properties:

— If a policy contains the edge (j, k'), then the cost of j is equal to the cost of m.

— If a policy contains the other edge from j, then

$$V(k') = \frac{(2^k-1)}{2^k}V(j) + \frac{V(m)}{2^k} \Rightarrow V(j) - V(k') = \frac{(V(j) - V(m))}{2^k} \leq \frac{1}{2^k}.$$

Hence the difference between the costs of j's children is at most $\frac{1}{2^k}$.

The gadget $g_{2(n-k)}$ is added between vertex k and each of its two neighbors of the basic graph, for every k, $1 < k < n$, and $g_{2(n-1)}$ is added only between vertex 1 and $1'$ (because vertex 1 is switched only once). Note that the number of new average vertices is polynomial $(2(n-1)^2)$ in the number of the min vertices n. Let the resulting graph be G_n.

Now suppose that the initial policy is the 0-vector ($S = S_n S_{n-1} \cdots S_1 = 00...0$). With respect to this policy, every min vertex has cost 1. The best policy is edge 0 for every min vertex except vertex 1, and edge 1 for that ($S = 00...01$). With this policy, every min vertex has cost 1/2. Although this policy differs by only one edge from the initial policy, the difference policy improvement algorithm takes 2^n-1 steps to find it.

We next describe how the algorithm performs for the graph for n equal to 2. Initially both min vertices 1 and 2 are switchable. The difference between the values of the neighbors of vertex 1 is 1/8, whereas the corresponding difference for vertex 2 is 1/4, so it is first switched. Vertex 1 is switched next, and then vertex 2. We have a total of $2^2-1 = 3$ switches.

The values $V(0')$, $V(1)$, $V(2)$, $V(3)$ can be expressed by the following expressions (if n is greater than or equal to 3):

$$V(0') = \frac{1}{2}(1 + V(n)) = \frac{1}{2} + \frac{V(n)}{2},$$

$$V(1) = S_1 V(1') + (1-S_1)V(0') = \frac{S_1}{2} + (1-S_1)\left(\frac{1}{2} + \frac{V(n)}{2}\right) = \frac{1}{2} + \frac{V(n)}{2} - \frac{S_1 V(n)}{2} = V(0') - \frac{S_1 V(n)}{2},$$

$$V(2') = \frac{1}{2}(V(1') + V(0')) = \frac{1}{2}\left(\frac{1}{2} + \frac{1}{2} + \frac{V(n)}{2}\right) = \frac{1}{2} + \frac{V(n)}{4},$$

$$V(2) = S_2 V(2') + (1-S_2)V(1) = V(1) - \frac{S_2 V(n)}{4} + \frac{S_1 S_2 V(n)}{2} = V(1) - \frac{S_2}{2}\left(\frac{1}{2} - S_1\right)V(n).$$

$$V(3) \cdots = \cdots V(2) - \frac{S_3}{2}\left(\frac{1}{2} - S_1\right)\left(\frac{1}{2} - S_2\right)V(n).$$

This leads us to Lemma 2.2 which is the first step of the proof that the difference policy improvement algorithm makes an exponential number of switches to reach the best policy for this structure. In the main lemma of this proof (Lemma 2.7) we show that if the policy for the leftmost k vertices is 0...00, and these

vertices are switchable, the next 2^k-1 switches of the algorithm are made on these vertices to reach the policy 0...01 for these vertices. The most important step to that lemma is to express the cost of every vertex with a formula that depends on the current policy and on costs of other vertices (Lemma 2.2). (After that point we do not need the graph.) The formulas give necessary and sufficient conditions (depending only on the current policy), for a vertex to be switchable (Corollary 2.4). The next two results (Corollary 2.5 and Lemma 2.6, which are also used in the proof of the main lemma) give relations between the switching of a vertex and other vertices becoming switchable.

Definition 2.1. $a(k)$, for every positive k, is defined to be

$$a(k+1) = a(k)\left(\frac{1}{2} - S_k\right) \text{ and } a(1) = -\frac{1}{2}.$$

The next lemma is technical and it is only used in the proof of Lemma 2.2.

Lemma 2.1. For every $2 \le k \le n$:

$$V(k') - V(k-1) = \frac{a(k)}{a(k-1)}\left(V((k-1)') - V(k-2)\right),$$

where $V(0)$ should be substituted by $V(0')$ in case k is 2.

Proof: Note that addition of the gadgets does not affect the costs that the vertices get, but only hides the actual difference between the min vertices.

$$V(k') - V(k-1) = \left(\frac{1}{2}V((k-1)') + \frac{1}{2}V(k-2)\right) - \left(S_{k-1}V((k-1)') + (1-S_{k-1})V(k-2)\right) \text{ (by the construction)} =$$

$$= \frac{a(k)}{a(k-1)}\left(V((k-1)') - V(k-2)\right) \text{ (by Definition 2.1.) } \square$$

Lemma 2.2. For every k, $2 \le k \le n$, the costs of the vertices, with respect to the current policy S, are given by the following formulas:

$$V(k) = V(k-1) + S_k V(n)a(k) \text{ and } V(1) = V(0') + S_1 V(n)a(1), \qquad \text{(I)}$$

$$V(k') = V(k-1) + V(n)a(k), \; V(1') = V(0') + V(n)a(1), \text{ and } V(0') = \frac{1}{2} + \frac{V(n)}{2}, \qquad \text{(II)}$$

where $a(k)$ is as in Definition 2.1.

Proof: The formulas can be proved by induction on k. The pair (I) is proved first:

Basis Case: It has been proved that (I) holds for k equal to 1, 2 and 3 by the relations presented before the lemma.

Induction Hypothesis: We suppose that (I) holds for every $k \le m$.

Induction Step: We prove that (I) holds for k equal to $m+1$.

$$V(m+1) = S_{m+1}V((m+1)') + (1-S_{m+1})V(m) \text{ (by the construction)} =$$

$$= V(m) + S_{m+1}\left(V((m+1)') - V(m-1) - S_m V(n)a(m)\right) \text{ (induction hypothesis)} =$$

$$= V(m) + S_{m+1}\left(\frac{1}{2}V(m') + \frac{1}{2}V(m-1) - V(m-1) - S_m V(n)a(m)\right) \text{ (by the construction)} =$$

$$= V(m) + S_{m+1}\left(\frac{1}{2}\frac{a(m)}{a(m-1)}(V((m-1)') - V(m-2)) - S_m V(n)a(m)\right) \text{ (by Lemma 2.1)} = \cdots =$$

$$= V(m) + S_{m+1}\left(\frac{1}{2}\frac{a(m)}{a(1)}(V(1') - V(0')) - S_m V(n)a(m)\right) \text{ (by applying Lemma 2.1 } m-2 \text{ times)} =$$

$$= V(m) + S_{m+1}V(n)a(m)\left(\frac{1}{2} - S_m\right) \text{ (by the construction)} =$$

$$= V(m) + S_{m+1}V(n)a(m+1) \text{ (by Definition 2.1).}$$

Proof of the pair (II):

Basis Case: The formulas obviously hold for k equal to 0 and 1. The basis case for k equal to 2 is derived as follows.

$$V(2') = V(1) + V(n)a(2) = V(0') + S_1 V(n)a(1) + V(n)a(1)\left(\frac{1}{2} - S_1\right) =$$

$$= \frac{1}{2} + \frac{V(n)}{2} + \frac{1}{2}V(n)a(1) = \frac{1}{2} + \frac{V(n)}{4}.$$

Induction Hypothesis: We suppose that (II) holds for every $k \le m$.

Induction Step: We prove that (II) holds for k equal to $m+1$.

$$V((m+1)') = \frac{1}{2}\left(V(m') + V(m-1)\right) \text{ (by the construction)} =$$

$$= V(m-1) + \frac{1}{2}V(n)a(m) \text{ (induction hypothesis)} =$$

$$= V(m) + \left(\frac{1}{2} - S_m\right)V(n)a(m) \text{ (by applying (I) for } m) =$$

$$= V(m) + V(n)a(m+1) \text{ (by Definition 2.1). } \square$$

From Definition 2.1, the following corollary is easily derived.

Corollary 2.3. If $a(k)$ is as Definition 2.1, then the following holds, for every positive k.

$$a(k+1) = \frac{(-1)^{S_k}}{2}a(k)$$

Proof: The proof is based on the recursive formula for $a(k+1)$: $a(k+1) = a(k)\left(\frac{1}{2} - S_k\right)$.

If S_k is equal to 1 then $a(k+1) = -\frac{1}{2}a(k) = \frac{(-1)^{S_k}}{2}a(k)$. If S_k is equal to 0 then $a(k+1) = \frac{1}{2}a(k) = \frac{(-1)^{S_k}}{2}a(k)$. □

Corollary 2.4. Vertex k is switchable if and only if either S_k is 0 and $a(k)$ is negative, or S_k is 1 and $a(k)$ is positive, for every positive k.

Proof: Proof that if k is switchable and S_k is 0, $a(k)$ is negative:
If k is greater than 1 then, according to Lemma 2.2,

$$V(k) = V(k-1) + S_k V(n)a(k).$$

Since S_k is 0, $V(k) = V(k-1)$. Since the switch of the vertex to S_k equal to 1 gives a smaller cost,

$$V(k-1) > V(k-1) + V(n)a(k) => a(k) < 0.$$

(since $V(n)$ is always positive). The rest can be proved by a very similar reasoning. □

Corollary 2.5. If vertex k (for every positive k) is switchable, and one or more larger numbered vertices are switched arbitrarily, vertex k is still switchable.

Proof: The fact that k is switchable means that either S_k is 0 and $a(k)$ is negative, or S_k is 1 and $a(k)$ is positive. Since the switches of larger numbered vertices do not have any effect on the cost of $a(k)$, k is still switchable after these switches. □

Lemma 2.6. If vertex k (for every positive k less than n) is switched, and $S_n \cdots S_{k+2}S_{k+1} = 0...01$, then all the vertices $k+1$, $k+2$, ..., n are switchable.

Proof: First, suppose that S_k is switched from 0 to 1. From Corollary 2.4, it is deduced that $a(k)$ is negative. By Corollary 2.3, $a(k+1)$ is positive, so by Corollary 2.4, vertex $k+1$ is switchable. By Corollary 2.3, $a(k+2)$ is negative, so by Corollary 2.4, vertex $k+2$ is switchable. It can be proved by induction that the other larger numbered vertices are switchable, too.
If S_k is switched from 1 to 0, the proof that the lemma holds is the same. □

Lemma 2.7. The following two statements hold (for every positive k):
— If $S_n \cdots S_{k+1}S_k = 0...01$, and the vertices k, $k+1$, .., n are switchable, the next $2^{n-k+1}-1$ switches of the difference policy improvement algorithm are made on these vertices, to reach the policy where $S_n \cdots S_{k+1}S_k = 0...00$.
— If $S_n \cdots S_{k+1}S_k = 0...00$, and the vertices k, $k+1$, .., n are switchable, the next $2^{n-k+1}-1$ switches of the difference policy improvement algorithm are made on these vertices, to reach the policy where $S_n \cdots S_{k+1}S_k = 0...01$.

Proof: The lemma will be proved by induction on k. Before that, an important notice on the difference between the costs of the vertices:

According to Lemma 2.2, the cost of vertex k ($k \leq n$), according to the policy $S = S_n S_{n-1} ... S_1$, is

$$V(k) = V(k-1) + S_k V(n) a(k) \text{ and } V(1) = V(0') + S_1 V(n) a(1).$$

From these formulas it follows that the difference between the costs of the two neighbors of vertex k is $V(n)a(k)$, or $\frac{V(n)}{2^k}$ (by Corollary 2.3), for the basic structure (without the gadgets). Note that the difference is constant for every vertex and does not depend on the policy. With the gadgets added to it, the difference for k becomes $\frac{V(n)}{2^{2n-k}}$, which is an increasing function of k, so the algorithm performs like the simple policy algorithm.

Basis Case: It is obvious that the statements hold for the case $k = n$.

Induction Hypothesis: The statements hold for every $k \geq m$.

Induction Step: We will prove that the statements hold for k equal to $m-1$. We prove the first one first:

The difference policy improvement algorithm works first on the vertices numbered n, $n-1$, ..., m, since the select procedure always chooses the highest numbered switchable vertex. From the second statement of the induction hypothesis, we deduce that it performs $2^{n-m+1}-1$ switches to reach the policy where $S_n \cdots S_{k+1} S_m = 0...01$. Vertex $m-1$ is still switchable (Corollary 2.5), so it is switched (from $S_{m-1} = 1$ to 0). By Lemma 2.6, the vertices n, $n-1$, ..., m are again all switchable. From the first statement of the induction hypothesis, we deduce that the difference policy improvement algorithm performs $2^{n-m+1}-1$ switches on these vertices to reach the policy where $S_n \cdots S_{k+1} S_m = 0...00$. The total number of the switches is $2^{n-m+2}-1$.

The second statement is proved following the same reasoning. □

We can now present the main result of the section.

Theorem 2.8. The difference policy improvement algorithm requires exponential time in the worst case.

Proof: Consider the graph G_n. For every positive n, the algorithm makes 2^n-1 switches to find the best policy (vector 00...01) if the initial policy is the 0-vector. This follows from Lemma 2.7, if initially every vertex is switchable. Since the initial policy is the 0-vector and $a(1)$ is negative, every other $a(k)$ is negative, too; so, by Corollary 2.4, every vertex is switchable. □

As corollaries of this theorem, it also follows that the simple and topological improvement algorithms require exponential time in the worst case. This is because on the graph G_n, the simple and topological policy improvement algorithms perform just like the difference algorithm.

Corollary 2.9. The topological policy improvement algorithm requires exponential time in the worst case.

Corollary 2.10. The simple policy improvement algorithm requires exponential time in the worst case.

The proof of Theorem 2.8 showed that the difference between between the costs of the two neighbors of a vertex is not proportional to the decrease that will actually result by making the switch. As a result, the difference algorithm can require exponential time. A better approach might be to compute for each switchable vertex the decrease that its switch will give, and decide which vertex to select based on that. This observation leads us to the best decrease policy improvement algorithm.

2.2. The Best Decrease Algorithm

The select procedure of this algorithm first topologically sorts the vertices of the graph. For every switchable min vertex in the component with the lowest order that contains switchable vertices, the decrease of the cost of the vertex if it is switched is computed. The vertex that will get the largest decrease is selected (if more than one vertex can get this decrease, the largest numbered one is selected).

The algorithm can find the best policy of the graph G_n in one step, if the initial policy is the 0-vector. However, we can construct another graph on which the best decrease algorithm requires exponential time. For the new structure we will use another gadget; the **gadget $\mathbf{g_{m,l}}$** is presented in Figure 2.4.

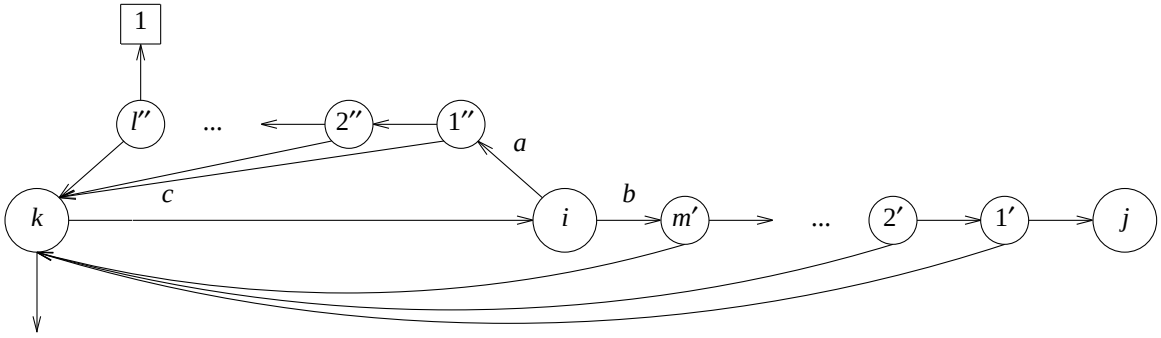

Figure 2.4: Gadget $g_{m,l}$

The gadget $g_{m,l}$ is placed between a min vertex k and another vertex j, that can be a min, average or sink vertex. The gadget introduces a new min vertex i, and $l+m$ new average vertices. Each vertex p' of the set of m average vertices, has one edge going to min vertex k and another going to the previous average vertex $(p-1)'$ or to vertex j if p is 1. Each vertex p'' of the set of l average vertices (double-primed vertices), has one edge going to min vertex k and another going to the next average vertex $(p+1)''$ or to the 1-sink if p is l. Finally, there are edges from the new min vertex i to the average vertices $1''$ and m', and an edge from min vertex k to min vertex i. The letters a, b and c denote the labels (0 or 1) of the edges $(i, 1'')$, (i, m') and (k, i), respectively.

Roughly, the key property of this gadget is as follows. If $S_k \neq c$, $S_i \neq b$ and $V(j)$ is much smaller than $V(k)$, then k and i can decrease in cost by a large amount if **both** are switched. However, if only one of k or i is switched, the decrease in cost of k or i is small.

It can be easily checked that the gadget has the following properties:

— If $S_k = c$ and $S_i = b$, then $V(k) = V(j)$.

— If $S_i = a$, then $V(i) = (1 - 2^{-l})V(k) + 2^{-l}$.

— If $S_i = b$, then $V(i) = (1 - 2^{-m})V(k) + 2^{-m}V(j)$.

— If $S_k \neq c$, then the improvement of $V(i)$ when it is switched from a to b is:

$$V(k) + 2^{-l}(1 - V(k)) - V(k) - 2^{-m}(V(j) - V(k)) = 2^{-m}(V(k) - V(j)) + 2^{-l}(1 - V(k)).$$

— If $S_k \neq c$, then the improvement of $V(i)$ when it is switched from b to a is:

$$V(k) + 2^{-m}(V(j) - V(k)) - V(k) - 2^{-l}(1 - V(k)) = 2^{-m}(V(j) - V(k)) - 2^{-l}(1 - V(k)).$$

The last two properties make sense only if the cost of vertices j and k are the same before and after the switch of vertex i, because in this case $V(j)$ and $V(k)$ are the same with respect to the current policy before and after the switch. This is true for the structure that will be presented because no edge (other than the edge labeled c in the gadget) will connect any vertex with vertex i.

3. Conclusion and Open Problems

We have studied the complexity of the policy improvement algorithm for Markov decision processes and have shown that many natural variations of this algorithm require exponential time in the worst case.

Apparently, no probabilistic analysis of the policy improvement algorithm has been undertaken, motivating the following questions. Suppose we define a new algorithm, the **Randomized Policy Improvement Algorithm**, by defining the select procedure to choose a switchable vertex randomly and uniformly from the set of switchable vertices. Does the randomized policy improvement algorithm run in expected polynomial time on all inputs? For all of the constructions in this paper, the randomized policy improvement algorithm requires only polynomial expected time (we leave it to the reader to verify that this is the case). It would also be interesting to obtain results on the average case performance of the policy improvement algorithm, on reasonable distributions of inputs. The work of Tovey [12], may be a useful start in this direction.

A select procedure widely used in practice is one in which all switchable states are switched at each iteration. Does this algorithm require exponential time in the worst case?

References

[1] A. Condon. The Complexity of Stochastic Games. *Information and Computation*, to appear, 1990. Also available as Technical Report Number 863, Computer Sciences Department, University of Wisconsin-Madison.

[2] C. Derman. *Finite State Markov Decision Processes.* Academic Press, 1972.

[3] J. Gill. The Computational Complexity of Probabilistic Turing Machines. *SIAM Journal on Computing*, 6:675-695, 1977.

[4] Howard. *Dynamic Programming and Markov Processes.* M.I.T. Press, 1960.

[5] R. J. Jeroslow. The Simplex Algorithm with the Pivot Rule of Maximizing Criterion Improvement. *Discrete Math.*, 4:367-378, 1973.

[6] D. S. Johnson, C. H. Papadimitriou and M. Yannakakis. How Easy is Local Search? *Journal on Computer and System Sciences*, 37:79-100, 1988.

[7] L. G. Khachiyan. A Polynomial algorithm in linear programming. *Soviet Math Dokl.*, 20:191-194, 1979.

[8] V. Klee and G. Minty. How Good is the Simplex Algorithm? *Inequalities III*, O. Shisha, Academic Press, New York, 159-175, 1979.

[9] C. H. Papadimitriou, A. A. Schäffer and M. Yannakakis. On the Complexity of Local Search. *Proceedings of the 22nd Annual Symposium on the Theory of Computing (STOC)*, 438-445, 1990.

[10] H. J. M. Peters and O. J. Vrieze. *Surveys in game theory and related topics, CWI Tract 39.* Centrum voor Wiskunde en Informatica, Amsterdam, 1987.

[11] L. S. Shapley. Stochastic Games. *Proceedings of the National Academy of Sciences, U.S.A,* 39: 1095-1100, 1953.

[12] C. A. Tovey. Low Order Polynomial Bounds on the Expected Performance of Local Improvement Algorithms. *Mathematical Programming*, 35(2): 193-224, 1986.