On Identifying Stable Ways to Configure Systems

Gagan Aggarwal [*] Mayur Datar [†] Nina Mishra [‡] Rajeev Motwani [§]

Abstract

We consider the often error-prone process of initially building and/or reconfiguring a computer system. We formulate a new optimization framework for capturing certain aspects of this system (re)configuration process. We describe offline and online algorithms that could aid operators in making decisions for how best to take actions on their computers so as to maintain the health of their systems.

1. Introduction

The purpose of this paper is to describe algorithms that enable operators to build and alter systems in ways that ensure their systems remain healthy and stable. Our efforts are directed towards helping operators since many studies suggest that the lack of sufficient knowledge on the part of operators tends to be the largest cause of system failures [7, 12, 5]. One study shows that over half of Internet service failures are due to operator error [12, 5], specifically hardware or software system misconfiguration. In addition, operators account for the largest fraction of the cost of systems (some figures indicate that operators cost 3 to 18 times more than the cost of the hardware itself [5]). Finally, operators have the single largest influence on time to recover from system failures [7, 12]. Consequently, our algorithms are directed towards helping operators initially build and also change their system configurations.

Despite the fact that operators are the largest cause of system failures, they are also the greatest source of knowledge regarding how to avoid system failures. Thus we focus on methods for enabling operators to benefit from the experiences of other operators so as to create more stable systems.

In order to accurately direct operators, systems need to be properly instrumented to collect data about operator actions and system health. Enriquez, Brown, and Patterson articulated the need for instrumentation by drawing on the experiences from the very reliable public telephone network system [4]. One of their findings was that the detailed logs collected by operators after each service failure was critical to understanding and improving reliability. In a related manner, we propose to collect data about other operators by recording more precise information about the health of a system after an operator action. In some cases, computer systems may be instrumented to automatically collect the needed data. In other cases, it may be necessary for an operator to manually record such information (as in the case of the telephone network). By collecting data about the health of a system after each operator action, we hope to create a weighted graph (described below) that we can then subsequently analyze in order to suggest healthy ways for operators to perform actions.

In order to motivate the generation of this weighted graph, we initiate a theoretical investigation into algorithms that can exploit this graph in order to suggest to operators safe ways to change the state of their system. The problems studied are action-based: Given a machine's current state s and given a sequence of actions, how can an operator find an effective way to execute those actions (by possibly performing suitable intermediate actions)? We formally define the problems in Section 1.3.

1.1. A Motivating Example

Prior to formally defining the problems, we begin with a sample scenario that is the source of many system problems: the system (re)configuration problem. Consider specifically the procedure a system administrator follows in order to initially build a computer system according to a users' requirements. The requirements typically take the following form: build a system with software X running stably, software Y supporting Z simultaneous users, etc. The operator translates these requirements into several subgoals.

[*] `gagan@cs.stanford.edu`. Department of Computer Science, Stanford University. Supported in part by a Stanford Graduate Fellowship, NSF Grant ITR-0331640 and NSF Grant EIA-0137761.

[†] `datar@cs.stanford.edu`. Google Inc. Supported in part by NSF Grant EIA-0137761.

[‡] `nina.mishra@cs.stanford.edu`. HP Labs and Stanford University. Supported in part by NSF Grant EIA-0137761.

[§] `rajeev@cs.stanford.edu`. Department of Computer Science, Stanford University. Supported in part by NSF Grants IIS-0118173, EIA-0137761, and ITR-0331640, an SNRC grant, and grants from Microsoft and Veritas.

1. Install Appropriate Drivers - to enable the operating system to properly communicate with the hardware
2. Install Level 1 Applications - these are widely-used applications where most bugs have already been identified
3. Install Level 2 Applications - these are less understood applications where some bugs have been identified
4. Install Level 3 Applications - these applications are largely unknown to the operator, and are also very rarely used.

The first subgoal is to install the appropriate drivers so that the operating system may properly communicate with the hardware. The order in which the drivers are installed is critical to assuring a healthy system. For a new piece of hardware, an operator may spend a substantial amount of time identifying the appropriate drivers and proper order in which to perform the installation. Since this path must be followed for all systems of a given hardware type, and since the success of this part of the installation is crucial to the remaining steps of the process, it is worth putting effort into finding the best path to the needed set of drivers. Indeed, it is often the case that operators document this path so that subsequent installations proceed smoothly to this state. This problem can be cast in a model that has previously been studied in the networking community and is mentioned in Section 2.

The second subgoal is to install Level 1 Applications, i.e., those with an extremely large user base where almost all bugs have already been fixed. Examples of such applications are Microsoft Office and Internet Explorer (IE). For this part of the process, the operator may use knowledge about previous installations to determine a good method for installing an ordered sequence of applications. In particular, an operator may seek to find a good way to first install IE and then install Office, since IE sets up network functionality used by Office. This problem is similar to the Offline Ordered problem defined in Section 1.3.

The third and fourth goals are to install Level 2 and Level 3 Applications. This set of applications are less understood by operators since they tend to have a smaller user base and don't necessarily have most bugs resolved. Furthermore, since fewer people request installation of Level 2 and Level 3 applications, there is less incentive for operators to perform these steps correctly. Thus our algorithms for the Offline Ordered problem should again be useful for this step.

Once a system has been successfully built, an operator may be subsequently called upon to install additional software in an online fashion. In such a setting, the start state is the boot state and the goal of the operator is to find a stable way to perform the application installation. Once an application has been installed, there may be a subsequent need to install another application. Thus the operator receives a sequence of online actions and must always find a stable way of performing these actions. This problem is similar to the Online Ordered problem defined in Section 1.3.

1.2. The Model

We formulate a new optimization framework for capturing certain aspects of the system (re)configuration problem. We model the problem as a directed, multigraph $G = (V, E)$. The vertices in V represent the set of possible states of a system. The *states* may be determined by any number of system configuration parameters, e.g., the hardware configuration (size of memory, type/speed of chip, amount of disk space, kernel parameters), the software configuration (installed patches, drivers, applications). There is also a designated *start state* whose interpretation varies depending on the problem being solved. In the event that a system is first being built/configured, the start state may simply be the hardware configuration. In the event that a system is being reconfigured, the start state may be the state of the system at "boot" time.

The edges in E correspond to different *actions* that can be performed on the system. Examples of actions include INSTALL DRIVER MDAC, INSTALL ORACLE 8I, INSTALL PATCH 29296, etc. Given that the system is in state u, the execution of an action corresponding to an edge (u, v), transforms the state of the system from u to v. (It is possible that in certain cases $u = v$, in which case the edge (u, v) corresponds to a self loop.) The colors represent possible actions and the color c_e of an edge $e = (u, v)$ corresponds to the action that transforms the system from state u to state v. Different edges may correspond to the same action and thus will have the same color. The edge-length $l(e)$ of an edge $e = (u, v)$ is the cost, in terms of system health, of executing the corresponding action in state u. For instance, we could associate a probability $p(e)$ with each edge $e = (u, v)$, which is the conditional probability that a particular software does not work reliably when the action corresponding to c_e is executed in state u. In that case, the conditional probability of success associated with a path is the product of the success probabilities along the edges in the path. To obtain additive path lengths, we then define the edge-lengths as the negative log of the success probability, i.e., $l(e) = -\log p(e)$. We allow multiple edges between two states, each possibly of different length and color, retaining only the shortest length edge for each color if there are multiple of the same color. In general, we do not assume that edge lengths are symmetric.

Note that the way in which the knowledge of other operators is utilized is in the edge lengths, since they indicate the likelihood a particular action will negatively influence the health of the system. In addition, the set of states is completely determined by how other operators have configured their systems. Thus while the total number of possible states is quite large, the total number of states reached by a large fraction of operators is likely to be significantly smaller.

1.3. Problem Statement and Results

We formulate and investigate specific Action-based optimization problems related to this graph. In the problems considered, the operator is given a current state of the system, s, and a collection of actions $c_1, \ldots, c_p$, specified in an ordered or unordered fashion, and if ordered, offline or online fashion. The goal is to find a safe way to execute those actions starting from state s. In the first variant, the actions are ordered and provided in an offline fashion.

Problem 1 (Offline Ordered Problem) *Given a graph $G = (V, E)$ with start state s and edge lengths $l(e)$ and given an offline sequence of actions $c_1, \ldots, c_p$, find a minimum length path starting from s that covers the actions in the given order. A path covers a particular action c_i if some edge on the path has that action.*

Note that the path may include intermediate edges that may or may not have the actions $c_1, \ldots, c_p$. We give a simple dynamic programming algorithm for optimally solving this problem.

In the next problem, the actions are again ordered, but now provided one at a time in an online fashion, i.e., one at a time after the previously-specified action has been performed.

Problem 2 (Online Ordered Problem) *Given a graph $G = (V, E)$ with start state s and edge lengths $l(e)$ and given an online sequence of actions $c_1, \ldots, c_p$, find a minimum length path starting from s that covers the actions in the given order.*

We show that a natural greedy algorithm has unbounded competitive ratio[1]. We give a $(2n - 1)$ deterministic lower bound for this problem and also a deterministic algorithm with an $O(|E|)$ competitive ratio. In addition, we show that a $\log^{O(1)} n$-competitive ratio can be obtained via a randomized algorithm.

Finally, we consider the problem where the actions are provided offline, but in an unordered fashion.

Problem 3 (Offline Unordered Problem) *Same as the Offline Ordered Problem, except that the colors are specified as a set $(\{c_1, c_2, \ldots, c_p\})$, rather than a sequence. The path may cover the colors in any order.*

1 The *competitive ratio* of an online algorithm (for a minimization problem) is the maximum over all input instances of the ratio of the cost of the solution produced by the online algorithm to the cost incurred by an optimal offline algorithm that knows the entire input sequence a priori.

We show this problem is hard to approximate.

2. Related Work

Metrical Task Systems. The Action-based problems are closely related to the Metrical Task System (MTS) problem (see Chapter 9 [1]). In an MTS instance, we are given an undirected graph $G = (V, E)$ (a metric space) and a set of tasks with a cost of executing each task in each state and a cost of changing states as edge lengths. Given a task sequence, the goal is to find a way of executing the tasks so that the sum of the total cost of executing the tasks and the total cost of changing states is minimized. Action-based problems and MTS are different to the extent that in an MTS, a task does not force a change of state and also each task can be executed in any state (although if the cost of executing a task in a state is infinite, the MTS can be forced to change state). In fact, many solutions to online MTS exploit the possibility of staying in a fixed state while executing multiple tasks [1] – something that is not possible in our case.

Shortest-Path Problems. One problem formulation that follows from Step 1 of the motivating example “Install Appropriate Drivers” is the following: Given a graph $G = (V, E)$ where each edge has a weight $l(e)$ and given a designated start and end state, find a minimum weight path from the start to the end state. The relationship between this problem and installing appropriate drivers is that an operator may know the current set of drivers and the final set of drivers (start and end state) and may just wish to find a path from the start to the end state so as to ensure that the health of the system can be maximized. We observe that this problem can be solved via Dijkstra’s shortest path algorithm.

Generalizing this further, one may consider the setting where each edge has a cost $l(e)$ and multiple weights $w_1(e), \ldots, w_k(e)$ and the goal is to find a minimum cost path p from the start to the end state subject to the constraint that the edge weights do not exceed a specified threshold, i.e., minimize $\sum_{e \in p} l(e)$ so that $\sum_{e \in p} w_i(e) \leq T_i$ for specified thresholds T_i, $i = 1, \ldots k$. Such a problem statement corresponds to the setting where an operator may wish to find a path that minimizes the amount of work required by the operator and yet also ensures that the probability software X works is greater than .95, the probability that software Y works is greater than .95, etc. This optimization problem has been studied in the context of network routing. When designing a network routing algorithm, many factors come into play: a feasible route must be discovered, e.g., one that satisfies bandwidth, delay, and jitter constraints while also efficiently utilizing network resources. The problem is known as the Multi-Constrained Optimal Path Problem. In the case that there is only one weight $w_1(e)$, the problem is known to have a polynomial-time approximation scheme [8]. In the case where each edge has multiple weights, algorithms are known that find a path with cost no larger than the optimum, but that exceed the specified thresholds by a small factor [6]. Heuristics have also been studied by Korkmaz and Krunz [9] as well as Yuan and Li [14]. Other results from the network routing literature can be found in [13, 10, 3].

3. Algorithms

In this section we describe our results for each of the three Action-based problems.

3.1. Offline Ordered Problem

This is the easiest variant of the Action-based problems. We present a simple offline algorithm that solves the problem optimally. The algorithm uses dynamic programming and is similar to the offline algorithm for MTS (see Section 9.4 [1]). For each state x, we define a variable $f(x,i)$ which represents the length of the shortest path from s to x that covers the first i colors. We set $f(s,0)=0$, and for $x \neq s$ we set $f(x,0)=l(s,x)$.[2] The following recurrence computes $f(x,i+1)$ assuming $f(v,i)$ has already been computed for all $v \in V$.

$$f(x,i+1) = \min_{\{e=(u,v)|e\in E\ \&\ c_e=c_{i+1}\}} f(u,i)+l(u,v)+l(v,x)$$

In addition to the preprocessing of computing all-pairs shortest paths, the running time of the algorithm is $O(n|E|p)$, where p is the length of the color sequence. Note that the additional colors on the intermediate edges correspond to suggested actions that the user should perform in order to sustain the system's health.

3.2. Online Ordered Problem

First we show that the natural greedy algorithm has unbounded competitive ratio. The greedy algorithm works as follows: On seeing a request for a color c in state x, if $(u,v) = \arg\min_{(u,v)\text{ has color }c}(l(x,u)+l(u,v))$, then the algorithm traverses edges (x,u) and (u,v). Consider the state space represented in Figure 1. Starting in state 1, if given a request sequence of colors $A,B,A,B,A,\ldots$, the optimal algorithm incurs a cost of $1+\epsilon$ while the greedy algorithm incurs a cost equal to the length of the sequence.

2 Throughout, we assume that the graph is complete, i.e., there is an edge between each pair of states. This can be achieved by adding in any missing edge (u,v) with length equal to that of the shortest path from u to v. The preprocessing involves computing all-pairs shortest paths in the graph G.

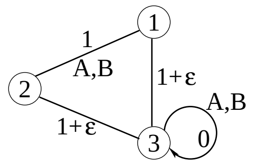

Figure 1. Counterexample for Greedy

As mentioned earlier, Action-based and MTS are closely related. An instance of MTS can be reduced to an instance of the Online Ordered Problem as follows: The graph $G=(V,E)$ for the Online Ordered Problem instance is the same as the metric space of the MTS instance. In addition, we add the following edges to G: for every pair of state x and task t, such that there is a finite cost $t(x)$ of executing the task in that state, we add a self-loop on x of length $t(x)$ and color t. This reduction implies that the Online Ordered Problem is at least as hard as MTS in terms of competitive ratio. Thus, the $(2n-1)$ lower bound for MTS [2, 11] also applies to the Online Ordered Problem.

We next give a reduction from the undirected case of the Online Ordered Problem to MTS that implies an $O(|E|)$ competitive ratio for undirected Online Ordered. (The argument will also apply to the directed case as we describe later.) In the undirected case, for each edge (u,v) there is also an edge (v,u) of the same length and color. Given an Online Ordered instance $G=(V,E)$, we define a metric space G' for MTS. At an intuitive level it is convenient to view the construction of G' from G as follows: make two copies of G (G_1, G_2) and for each of the copies, place an imaginary node in the middle of every edge, i.e., for each edge $e \in G$, create a node $e_1 \in G_1$ and $e_2 \in G_2$. Additionally "join" the two copies G_1, G_2, by connecting all pairs of nodes corresponding to the same edge e in G (e_1, e_2) with edges of length $l(e)$ respectively. These imaginary nodes are the nodes (states) in G'. For adjacent edges e, e' in G, we will also introduce appropriate edges connecting e_1, e'_1, e_2, e'_2.

Formally, the state set V' of G' consists of $V' = V_1 \bigcup V_2$, where $V_1 = V_2 = E$. Thus, there is a state $e_1 \in V_1$ (respectively, $e_2 \in V_2$) corresponding to each edge $e \in E$. For every pair of nodes e_1, e_2 corresponding to the edge e in G, we add the edge (e_1, e_2) of length $l(e)$. In addition, for every ordered pair of *adjacent* edges $e=(x,y)$ and $e'=(y,z)$ in E, there are four edges $(e_1,e'_1),(e_1,e'_2),(e_2,e'_1)$ and (e_2,e'_2) in G', each of length $\frac{l(e)+l(e')}{2}$. It can be shown that G' is indeed a metric, assuming that all edge lengths are nonnegative. Corresponding to each color c, we define two tasks c_1 and c_2 as follows: The task c_1 (respectively, c_2) when executed in any node in V_1 (respectively, V_2) that corresponds to an edge of color c has cost zero, but has cost ∞ on all other nodes. Finally, for any sequence of requests $a, b, c, d, e, \ldots$ in Online Ordered, we introduce a sequence of task requests in MTS such that the odd tasks $a, c, e, \ldots$ generate requests $a_1, c_1, e_1, \ldots$, while the even tasks $b, d, f, \ldots$ generate requests $b_2, d_2, f_2, \ldots$. Let e^s be any edge in E that is incident onto the start state s in V; then, we choose e^s_1 as the start state for MTS. Let D be the largest edge-length in E.

Lemma 3.1 *Given any path P in $G = (V, E)$ starting at state s, and covering a sequence of colors $a, b, c, d, e, \ldots$, there exists a path P' in G' of length at most the length of P plus D. Moreover, the path P' visits a sequence of states $a'_1, b'_2, c'_1, d'_2, \ldots$ in G', where the cost of executing tasks $a_1, b_2, c_1, d_2, \ldots$, respectively is 0.*

Proof: Consider the path P. Let $e^1, e^2, \ldots, e^m$ denote the sequence of edges in this path. The path P' is constructed as follows: It starts from start state e^0_1 in G', and then traces the edges $e^1, e^2, \ldots, e^m$ in P by the corresponding states, $e^i_{1,2}$ (denotes e^i_1 or e^i_2 depending on the current state) corresponding to e^i, in G'. It "jumps across copies" G_1, G_2 of the state space, whenever required, to ensure that whenever the path P picks a new color, path P' is in a state where the cost of the corresponding action is 0. More precisely, if edge e^i in path P picks a new color (in the sequence $a, b, c, d, \ldots$), then corresponding to e^i, path P' will contain the state $e^i_{(i+1) mod 2+1}$, where the cost of that action is 0. This guarantees that corresponding to the color sequence $a, b, c, d, e, \ldots$ in P, path P' visits a sequence of states $a'_1, b'_2, c'_1, d'_2, \ldots$ in G', where the cost of executing tasks $a_1, b_2, c_1, d_2, \ldots$, respectively is 0. In addition, the path P' goes from e^i_1 to e^{i+1}_2 or from e^i_2 to e^{i+1}_1 when the same edge occurs consecutively in the edge sequence $e^1, e^2, \ldots, e^m$, i.e. $e^i = e^{i+1}$.

Since the path P starts from the start state s, the first edge e^1 is incident on s. By construction of G', the edge e^0 corresponding to the start state e^0_1 in G' is incident on s. Thus, they are adjacent[3]. Moreover, since P is a path in G any two consecutive edges in the sequence $e^1, e^2, \ldots, e^m$ are adjacent. Thus, by construction of G' the cost of going from $e^i_{1,2}$ to $e^{i+1}_{1,2}$, corresponding to the pair e^i, e^{i+1} in P, is equal to $l(e^i)/2 + l(e^{i+1})/2$. Moreover the first transition, if it exists, is e^0_1 to $e^1_{1,2}$ with cost $(l(e_0) + l(e_1))/2$. Thus the total cost of the path P' is equal to $l(e_0)/2 + l(e_1) + l(e_2) + \ldots + l(e_{m-1}) + l(e_m)/2$ which is at most $l(e_1) + l(e_2) + \ldots + l(e_m) + D$, where D as defined earlier is the length of the longest edge in E. □

Lemma 3.2 *Let P' be any path in G' that starts in e^s_1 and* executes *a sequence of tasks $a_1, b_2, c_1, d_2, \ldots$, i.e., visits a sequence of states $a'_1, b'_2, c'_1, d'_2, \ldots$, where the cost of executing the tasks $a_1, b_2, c_1, d_2, \ldots$ is respectively zero. We can translate P' into a path P in G, starting in state s, and of length at most twice the length of P' plus D, such that it covers the sequence of colors $a, b, c, d, \ldots$.*

Proof: Let $e^s_1 = e^0_1, e^1_{j_1}, e^2_{j_2}, \ldots, e^m_{j_m}$ denote the sequence of states in path P', where each j_i is either 1 or 2. The sequence of tasks given as input to the MTS forces any finite solution to alternately visit states in V_1 and V_2 to execute the given sequence of tasks. This precludes any solution that involves staying in the same state. Thus, no two adjacent states in the sequence can be identical. The path P is constructed as follows: For every state $e^i_{j_i}$ in P', path P crosses the corresponding edge e^i in G, sometimes crossing it twice. By construction, every pair of adjacent states in the sequence $e^0_1, e^1_{j_1}, e^2_{j_2}, \ldots, e^m_{j_m}$ correspond to adjacent edges in G. Consider an adjacent pair $e^i_{j_i}, e^{i+1}_{j_{i+1}}$. The path P crosses e^i corresponding to $e^i_{j_i}$. After crossing e^i, it ends up on one of the two nodes corresponding to the edge e^i. If this is the appropriate node to cross e^{i+1}, i.e. e^{i+1} is also incident on this node, then it crosses e^{i+1} corresponding to $e^{i+1}_{j_{i+1}}$. Otherwise, if after crossing e^i it ends up on the wrong node, it has to cross e^i one more time in order to be on the appropriate node to cross e^{i+1}. For the first state e^0_1 in the sequence, note that it corresponds to an edge e^0, adjacent to the start state s in G. Thus, the cost of path P in G is at most $2(l(e^0) + l(e^1) + \ldots + l(e^m))$. On the other hand, since no two adjacent states in the sequence $e^0_1, e^1_{j_1}, e^2_{j_2}, \ldots, e^m_{j_m}$ are identical, the cost of path P' is at least $l(e^0)/2 + l(e^1) + l(e^2) + \ldots + l(e^{m-1}) + l(e^m)/2$. Thus, the cost of path P is at most twice the length of P' plus D. □

The reduction described above is approximation-preserving, i.e., if we have an online algorithm for MTS with competitive ratio k, we can use it to solve the undirected case of Online Ordered with competitive ratio $2k$. The Work Function Algorithm (WFA) ([2] and Section 9.4 [1]) can be used to solve the online MTS with competitive ratio of $2n' - 1$ where n' is the number of states in the metric space. Since the reduced instance has $2|E|$ states, we have a competitive ratio of $8|E| - 2$ for the undirected case of Online Ordered.

The above reduction also works for the general case of Online Ordered when $G = (V, E)$ is a directed graph, although the reduced MTS instance $G' = (V', E')$ is no more a metric, in that the edge lengths are not symmetric although they do satisfy the triangle inequality. The Work Function Algorithm (WFA) is $2n' - 1$ competitive for such an asymmetric case of MTS as well (Section 9.7 [1]), thus giving a $8|E| - 2$ competitive algorithm for the general Online Ordered problem.

While our discussion so far has been strictly about deterministic algorithms, by virtue of the fact that we have reduced our problem to an MTS problem, we note that the

3 They could be identical in which case path P' starts at e^1_1.

competitive ratio can be improved to $\log^{O(1)} n$ via randomized algorithms for MTS [1].

To summarize, we have proved the following:

Theorem 3.1 *There is a deterministic (resp., randomized) algorithm that solves the Online Ordered Problem with an $O(|E|)$ (resp., $\log^{O(1)} n$) competitive ratio. There is no deterministic algorithm that can achieve a competitive ratio better than $2n - 1$.*

3.3. Offline Unordered Problem

The following reduction shows that it is NP-hard to approximate the Offline Unordered problem to within any bounded factor: Consider a SAT formula with n variables $(x_1, \ldots, x_n)$ and m clauses $(C_1, \ldots, C_m)$. Construct a graph with $n + 1$ primary states $v_0, v_1, \ldots, v_n$. Between each pair of states v_{i-1} and v_i, for $1 \leq i \leq n$, there are two vertex disjoint (induced) paths, whose intermediate states are disjoint with respect to all other paths. For each clause C_j containing the variable x_i, there is an edge colored j in the first path. For each clause C_j containing the negation $\overline{x_i}$, there is an edge colored j in the second path. The total length of each path is 1, with the length being uniformly divided amongst the edges in the path. Thus, traversing the first path corresponds to setting $x_i = 1$ and traversing the second path corresponds to setting $x_i = 0$. The request set of colors is the set $\{1, \ldots, m\}$ which corresponds to satisfying all clauses. A feasible path of length at most n covering all the requested colors gives a satisfying truth assignment for the SAT formula, and, similarly, a satisfying truth assignment defines a feasible path of length at most n. If there is no satisfying assignment, there is no path that covers all the requested colors.

4. Future Work

Our efforts have so far only been directed towards developing theoretically sound algorithms for solving certain action-based problems. We do recognize that there are research obstacles to overcome in order to create the directed graph that we assume we are given: collecting enough data to have probabilities on edges will be difficult given that systems are currently not instrumented to collect data at the granularity we require. Given that a system is in some state, our algorithms assume that we know how the health of a system is affected after an operator action. As system collection tools continue to mature, we believe that this type of data will be available [4]. In addition, the number of possible states of a computer system is extremely large. More research will have to be done to understand how the state space can be reduced to a manageable size. This problem can best be overcome once real data is available.

From an algorithmic perspective, many interesting questions also remain. Can our algorithms be extended to handle the situation where there is a probability distribution over the next state? For the Online Ordered Problem, can we close the gap between the lower bound $(2n - 1)$ and the upper bound $(8|E| - 2 = O(n^2))$ on the competitive ratio of a deterministic online algorithm?

5. Acknowledgments

We thank Alan Lau for providing motivating examples for the problem.

References

[1] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[2] A. Borodin, N. Linial, and M. Saks. An optimal on-line algorithm for metrical task system. *Journal of the ACM*, 39(4):745–763, Oct. 1992.

[3] S. Chen and K. Nahrstedt. On fi nding multi-constrained paths. In *Proc. of the IEEE International Conference on Communications*, pages 874 –879, 1998.

[4] P. Enriquez, A. Brown, and D. Patterson. Lessons from the pstn for dependable computing. *Workshop on Self-Healing, Adaptive, and Self-Managed systems*, 2002.

[5] A. Fox and D. Patterson. Self-repairing computers. *Scientifi c American*, June 2003.

[6] A. Goel, K. G. Ramakrishnan, D. Kataria, and D. Logothetis. Effi cient computation of delay-sensitive routes from one source to all destinations. In *INFOCOM*, pages 854–858, 2001.

[7] J. Gray. Why do computers stop and what can be done about it? *Proc. of the 5th Symposium on Reliablity in Distributed Software and Database systems*, Jan. 1986.

[8] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations Research*, 17:36–42, 1992.

[9] T. Korkmaz and M. Krunz. Multi-constrained optimal path selection. In *INFOCOM*, pages 834–843, 2001.

[10] T. Korkmaz, M. Krunz, and S. Tragoudas. An effi cient algorithm for fi nding a path subject to two additive constraints. In *Measurement and Modeling of Computer Systems*, pages 318–327, 2000.

[11] M. Manasse, L. McGeoch, and D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, June 1990.

[12] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it? *4th USENIX Symposium on Internet Technologies and Systems*, Mar. 2003.

[13] P. Paul and S. Raghavan. Survey of qos routing. In *Proc. of the 15th Intl Conference on Computer Communication*, 2002.

[14] X. Yuan and X. Liu. Heuristic algorithms for multi-constrained quality of service routing. In *INFOCOM*, pages 844–853, 2001.