The Influence of Domain Properties on the Performance of Real-Time Search Algorithms

Sven Koenig and Reid G. Simmons

August 1996
CMU-CS-96-115

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations.

Keywords: domain properties (here: Eulerian graphs); efficient and inefficient real-time search algorithms (edge counting, Learning Real-Time A*, node counting); testbeds (blocksworld, eight puzzle, gridworld, quicksand state space, racetrack domain, reset state space).

Abstract

In this report, we investigate the influence of domain properties on the performance of real-time search algorithms. We study uninformed, revolving real-time search algorithms with minimal lookahead that solve suboptimal search problems, for example variants of Korf's LRTA* algorithm and edge counting (these algorithms have been used successfully in the literature). We demonstrate, both theoretically and experimentally, that they can search Eulerian domains (a superset of undirected domains) very easily: even the real-time search algorithms that can be intractable are always efficient in Eulerian domains. Because traditional real-time search testbeds (such as the eight puzzle and gridworlds) are Eulerian, they cannot be used to distinguish between efficient and inefficient real-time search algorithms. It follows that one has to use non-Eulerian domains to demonstrate the superiority of a real-time search algorithm across a wide range of domains – the studied real-time search algorithms differ in this respect from traditional search algorithms. To this end, we describe two classes of domains ("reset state spaces" and "quicksand state spaces") that do not suffer as much from the problems of the standard test domains and demonstrate the performance of various real-time search algorithms in them.

1 Introduction

Real-time heuristic search algorithms, a term coined by Korf [Korf, 1990], interleave search with action execution by limiting the amount of deliberation performed between action executions. [Korf, 1990] and [Korf, 1993] demonstrated that real-time search algorithms are powerful search algorithms that often outperform more traditional search algorithms. Empirical results for real-time search algorithms have typically been reported for AI search domains such as

- sliding tile puzzles (such as the 8-puzzle) [Korf, 1987], [Korf, 1988], [Korf, 1990], [Russell and Wefald, 1991], [Knight, 1993], [Korf, 1993], [Ishida, 1995]; and

- gridworlds [Korf, 1990], [Pirzadeh and Snyder, 1990], [Ishida and Korf, 1991], [Ishida, 1992], [Pemberton and Korf, 1992], [Thrun, 1992], [Matsubara and Ishida, 1994], [Stentz, 1995], [Ishida, 1995].

Prototypical test domains (such as these AI search domains) allow engineers to compare different search methods, which enables them to evaluate the algorithms without having to implement them. Often engineers have to generalize the results that have been reported in the literature to different search problems. It is therefore important that the performance of real-time search algorithms in the test domains be representative of their performance in the domains of interest: test domains should either reflect the properties of the domains of interest or, at least, be representative of a wide range of domains.[1] To this end, one has to understand how properties of domains affect the performance of real-time search algorithms.

Although researchers have studied which factors influence the performance of traditional search algorithms (such as the A* algorithm) [Pearl, 1985], currently not much is known about how domain properties influence the performance of real-time search algorithms. We investigate two classes of domains: a domain is considered a bad testbed for real-time search algorithms (Type 1) if no real-time search algorithm has a significant performance advantage over other (reasonable) real-time search algorithms, otherwise the domain is a good testbed (Type 2). In Figure 1, for example, state spaces of Type 1a are bad testbeds, because they are too hard to search with real-time search algorithms: even the most efficient real-time search algorithms perform poorly in them. Likewise, state spaces of Type 1b are bad testbeds, because they are too easy to search: even inefficient (but "reasonable") real-time search algorithms perform very well in them. State spaces of Type 2, on the other hand, are good testbeds, because they are better able to discriminate between good and bad real-time search algorithms.

Similar domains can be of different types. Consider, for example, the following two extremely simple blocksworld domains. The simplicity of these artificially constructed

[1] We do not argue whether micro worlds (often called "toy problems") should be used to evaluate AI algorithms experimentally, instead of complex (real-world) domains, see for example the discussion in [Hanks *et al.*, 1993]. However, we do argue that *any* kind of test domain should be carefully chosen if one does not want to compare different algorithms directly in the domains of interest.

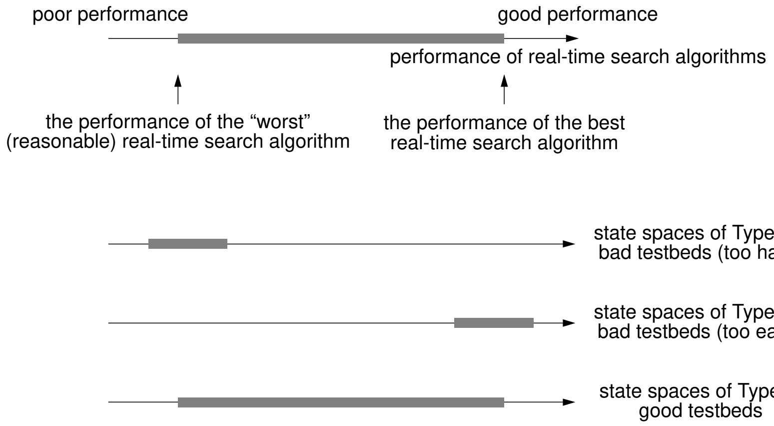

Figure 1: Good and bad testbeds (1)

domains allows us to understand the principles that underlie more realistic domains: In both cases, there are x indistinguishable blocks, all of which are initially on the table. The task is to stack all of them on top of one another on a platform, see Figure 2. Domain 1 has four operators: "pickup block from table," "put block on stack," "pickup block from stack," and "put block on table" (these are the operators available in standard blocksworld domains, expect that a pair of "pickup" and a "putdown" operators are usually merged into one atomic "move" operator). A block picked up from the table is always followed by a "put on stack," and a block picked up from the stack is always subsequently placed on the table. Domain 2 has the same two pickup operators and the same "put block on stack" operator, but the "put block on table" operator (which always follows a "pickup block from stack" operator) knocks down the whole stack onto the table. The state spaces of these two domains are very similar, see Figures 3 and 4: both have $3x + 1$ states, $4x$ actions, and the largest goal distance is $2x + 1$. Furthermore, corresponding states have the same small number of actions available, either one or two. Nevertheless, we will show that Domain 1 is of Type 1 for the real-time search algorithms that we study, while Domain 2 is of Type 2.

Which real-time search algorithm to use is not crucial when searching domains of Type 1. Assume, however, that one has to decide with which real-time search algorithm to solve some search problem in a domain of Type 2, but one only has an empirical evaluation of the real-time search algorithms in a domain of Type 1 available. In this case, the reported results appear to suggest that all real-time search algorithms perform equally well and that it does not matter much which of them is used to solve the problem. In reality, however, there is a huge difference in performance and one should carefully choose a suitable real-time search algorithm. If one would understand how to distinguish between domains of different types, authors of performance studies could publish performance results not only for domains of Type 1, but also for domains of Type 2, and readers of these empirical evaluations would be prevented from making wrong generalizations.

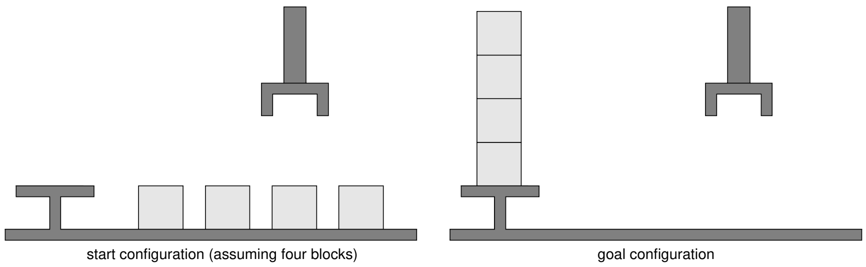

Figure 2: A simple blocksworld problem

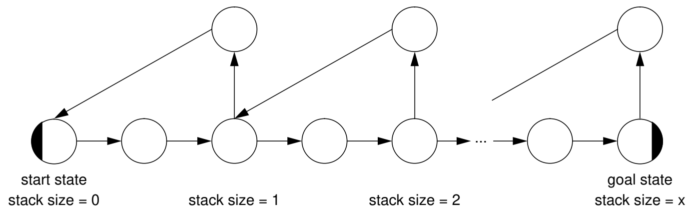

Figure 3: Domain 1

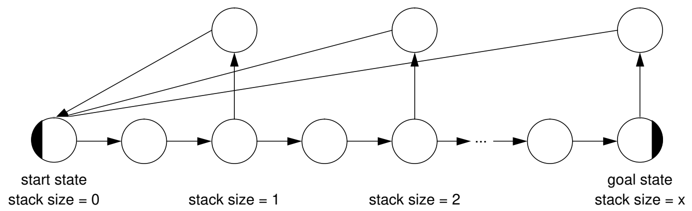

Figure 4: Domain 2

In this report, we make a first step in this direction: we study only one property of state spaces and we consider its effect only on the performance of selected real-time search algorithms, but not on a whole class of algorithms. We identify one particular property of domains that makes them bad testbeds for the studied real-time search algorithms, namely being Eulerian. In Eulerian state spaces, each state has an equal number of actions that enter and leave the state. All undirected domains are, for example, Eulerian.

We study several uninformed, revolving real-time search algorithms with minimal lookahead that solve suboptimal search problems. This restriction allows us to study the influence of domain properties on the performance of real-time search algorithms in isolation. For example, there is no difference in the kind and amount of domain knowledge that the algorithms have available and how they utilize it. We compare a variant of Korf's LRTA* algorithm – probably the best known real-time search algorithm – to other real-time search algorithms that have been used in the literature, both for Eulerian and non-Eulerian domains. Our analysis shows that Eulerian domains are all of Type 1b – even real-time search algorithms that are inefficient, in general, perform well in Eulerian domains. (Note that the Eulerian property has no effect on the performance of traditional search algorithms.) To confirm the practical applicability of these results, we supplement our theoretical worst-case analysis with an experimental average-case analysis that yields the same results.

Since sliding tile puzzles and gridworlds are typically undirected and therefore Eulerian, real-time search algorithms that can be intractable perform efficiently in such domains and have indeed been demonstrated successfully in such domains in the literature. Therefore, these domains are not appropriate to demonstrate how well real-time search algorithms perform in general. To remedy this, we propose two classes of non-Eulerian testbeds ("reset state spaces" and "quicksand state spaces") that are of Type 2. Therefore, they do not suffer as much from the problems of the standard test domains.

2 Notation

We use the following notation to describe state spaces formally: S denotes the finite set of states ("vertices") of the state space ("directed graph"), G with $\emptyset \neq G \subseteq S$ the non-empty set of goal states, and $s_{start} \in S$ the start state. $A(s)$ is the finite set of deterministic actions ("directed edges") that can be executed in $s \in S$ ("be traversed from s"), and $succ(s, a)$ denotes the uniquely determined successor state that results from the execution of action $a \in A(s)$ in $s \in S$.

The size of the state space is $n := |S|$, and the total number of state-action pairs (loosely called actions) is $e := \sum_{s \in S} |A(s)|$. $gd(s)$ denotes the goal distance of $s \in S$ (measured in action executions). Finally, the depth d of the state space (its "diameter") is its largest goal distance, $d := \max_{s \in S} gd(s)$.

There exist state spaces in which all of our real-time search algorithms can get trapped in a part of the state space that does not contain a goal state. To exclude these state spaces, we assume $d < \infty$, which implies $d \leq n - 1$. This means that, no matter which actions a real-time search algorithm has executed in the past, it can still reach a goal state. Strongly connected state spaces, for example, have this property. To simplify our descriptions, we also assume that $e \leq n^2$ (an extremely realistic assumption), since this allows us to state all complexity results in terms of n only. (Otherwise this assumption is unnecessary.)

The real-time search algorithm starts in state s_{start}. Initially, $memory = 0$ and $V(s,a) = 0$ for all $s \in S$ and $a \in A(s)$.

1. $s :=$ the current state.
2. If $s \in G$, then stop successfully.
3. Choose an a from $A(s)$ possibly using $memory$ and $V(s,a')$ for $a' \in A(s)$.
4. Update $memory$ and $V(s,a)$ possibly using $memory$, $V(s,a)$, and $V(succ(s,a),a')$ for $a' \in A(succ(s,a))$.
5. Execute action a, i.e. change the current state to $succ(s,a)$.
6. Go to 1.

Figure 5: Skeleton of the studied real-time search algorithms

3 Our Performance Measure

Real-time search algorithms differ from traditional search algorithms, in that they always maintain a current state. This is a state of the state space; it can only be changed by executing actions. We study suboptimal search – the task that real-time search algorithms can perform very well. Suboptimal search means looking for any path (i.e. sequence of actions) from the start state to a goal state. The sequence of actions that real-time search algorithms execute is such a path, although not necessarily an optimal one. In real-time search, the search time is (roughly) proportional to the length of the solution path, i.e. the total number of executed actions. Thus, we use the length of the path to evaluate the performance of real-time search algorithms.

When we refer to the complexity of a real-time search algorithm, we mean an upper bound on the total number of actions that it executes until it reaches a goal state, in big-O notation. This bound must hold for all possible topologies of state spaces of a given size, start and goal states, and tie breaking rules among indistinguishable actions. Two algorithms have the same tight complexity if the ratio of their worst-case performance is bounded from above and below by positive constants, although these constants can potentially be large.

4 Skeleton of Real-Time Search Algorithms

To make meaningful comparisons, one should only compare algorithms that make similar assumptions. We therefore restrict our attention to uninformed revolving real-time search algorithms with minimal lookahead (search horizon) and greedy action selection.

The algorithms maintain information in form of integer values $V(s,a)$, which are associated with every state-action pair (s,a). An additional integer value is maintained across action executions (in the variable *memory*). The semantics of these values depend on the specific real-time search algorithm used, but all values are zero-initialized, reflecting that the algorithms are initially uninformed. At no point in time can these values contain much information, since the algorithms must be able to decide quickly which actions to execute, and their decisions are based on these values. This requirement prevents the algorithms, for example, from encoding significant portions of the state space in these values.

The algorithms that we consider all fit the skeleton shown in Figure 5. They consist of a termination checking step (line 2), an action selection step (line 3), a value update step (line 4), and an action execution step (line 5). First, they check whether they have already reached a goal state and thus can terminate successfully (line 2). If not, they decide on the action to execute next (line 3). For this decision, they can consult the value stored in their memory and the values $V(s, a)$ associated with the actions in their current states. Then, they update the value of the selected action and their memory, possibly also using the values associated with the actions in their new state (line 4). Finally, they execute the selected action (line 5) and iterate this procedure (line 6).

All algorithms that fit this skeleton are easy to implement, uninformed, revolving real-time search algorithms with minimal lookahead and greedy action selection, although their action selection and value update steps can differ. They are uninformed, because they do not have any initial knowledge of the state space, not even estimates for the goal distances. They are revolving, because they repeat the same planning procedure after every action execution. They have minimal lookahead, because they use only information local to their current state in order to select an action. (They do not even have to project one action execution ahead.) Finally, they select only the action to execute next and do this with a minimal amount of search. This greedy way of selecting actions minimizes the amount of computation between action executions.

5 Worst-Case Results

We first study the complexity of real-time search algorithms over all state spaces. In this case, one can freely choose the state space that maximizes the number of action executions of a given real-time search algorithm (the "worst" state space) from all state spaces with the same number of states. Later, we restrict the possible choices and study the complexity of real-time search algorithms over a subset of all state spaces.

Empirical researchers sometimes consider complexity analyses to be unimportant, because they are more interested in the average-case performance of algorithms than in their worst-case performance. The reason why worst-case results are interesting is because such results provide performance guarantees. A proof that an algorithm has a small (worst-case) complexity over all state spaces, for example, guarantees that the algorithm is always efficient. This means that there cannot be any unpleasant surprises in form of unanticipated domains in which the performance of the algorithm

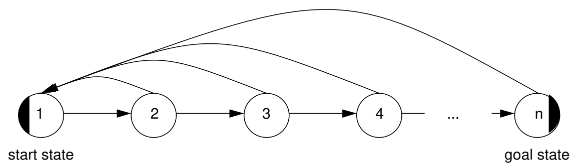

Figure 6: A simple reset state space

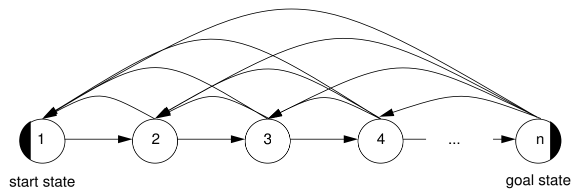

Figure 7: A more complex reset state space

deteriorates completely.

We are interested in the complexity of both efficient and inefficient real-time search algorithms, because the smaller the difference in the two complexities, the stronger the indication that search problems in such state spaces are of Type 1. (Experimental average-case results are provided in Chapter 6.)

5.1 General State Spaces

In this section, we introduce several algorithms that fit our real-time search skeleton (Figure 5) and consider their complexity in all state spaces with $d < \infty$ that have the same size n. In particular, we study a variant of a popular real-time search algorithm (namely min-LRTA*, a variant of Korf's LRTA* algorithm) and compare its complexity to the most efficient and less efficient real-time search algorithms.

Particularly tough state spaces to search are "reset" state spaces. A simple reset state space is a state space in which all states (but the start state) have an action that leads back to the start state (we say that the action "resets" the algorithms to the start state) – or, more generally, in which the lookahead of the real-time search algorithm is not large enough to avoid the execution of such actions. An example of a simple reset state space is shown in Figure 6. It is similar to our second blocksworld state space (Domain 2), shown in Figure 4. The reason why it is tricky to search reset state spaces is that the algorithms have to choose the correct action (out of two possible actions) $n-2$ times in a row in order to reach the goal state. If they execute only one action that is not on the optimal path from the start state to the goal state, they end up at the start state again and have to start all over.

A more complex reset state space is shown in Figure 7. It shares with simple reset state spaces the property that real-time search algorithms have to execute $O(n)$ actions on average to make up for a single mistake (to be precise: the average number of action executions is $(n-1)/2$ for simple reset state spaces and $n/3$ for more complex reset state spaces ($n \geq 3$) if one averages uniformly over all reset actions in non-goal states). However, since the more complex reset state space has $O(n^2)$ instead of $O(n)$ actions, the algorithms now have to choose the one correct action in every state out of $O(n)$ actions on average. The state space of Figure 7 is similar to state spaces of blocksworlds in which one can stack only single blocks, but (different from the blocksworld domains discussed earlier) one can remove an arbitrary number of blocks from the top of the stack.

5.1.1 LRTA*-Type Real-Time Search

Korf's Learning Real-Time A* (LRTA*) algorithm [Korf, 1987; Korf, 1988; Korf, 1990] can be used to find suboptimal and optimal solution paths. It is probably the most popular real-time search algorithm. Variants of LRTA* have, for example, been used by [Knight, 1993], [Ishida, 1995], and [Koenig and Simmons, 1995]. The variant that we use here is closely related to Q-learning, a widely-used reinforcement learning method, see [Koenig and Simmons, 1992]. We call it LRTA* with minimalistic lookahead (short: min-LRTA*), because the search horizon of its action selection step is even smaller than that of LRTA* with lookahead one. (We analyze Korf's original version of LRTA* with lookahead one in Section 9.)

The following table presents the action selection step and value update step of min-LRTA*. We use two operators with the following semantics: Given a set X, one-of X returns one element of X according to an arbitrary rule. (A subsequent invocation of one-of X can return a different element.) $\arg\min_{x \in X} f(x)$ returns the set $\{x \in X : f(x) = \min_{x' \in X} f(x')\}$.

Min-LRTA*	
action selection step (line 3)	$a :=$ one-of $\arg\min_{a' \in A(s)} V(s, a')$
value update step (line 4)	$V(s, a) := 1 + \min_{a' \in A(succ(s,a))} V(succ(s, a), a')$

The action selection step selects the state-action pair with the smallest value. The value update step replaces $V(s, a)$ with the more accurate lookahead value $1 + \min_{a' \in A(succ(s,a))} V(succ(s, a), a')$. This can be explained as follows: The value of any state-action pair $V(s, a)$ is always a lower bound on the number of action executions that one has to execute in order to reach a state with an unexplored action if one starts by executing action a in state s (unexplored actions can potentially lead to goal states). Thus, the number of action executions that one has to execute in order to reach a state with an unexplored action from state s' is at least $\min_{a' \in A(s')} V(s', a')$, and the number of action executions that one has to execute in order to reach a state with an unexplored action if one starts by executing action a in state s is at least $1 + \min_{a' \in A(succ(s,a))} V(succ(s,a), a')$, see [Koenig and Simmons, 1996] for details.

Min-LRTA* always reaches a goal state with a finite number of action executions. We can prove the following complexity result for min-LRTA*.

Theorem 1 *Min-LRTA* has a tight complexity of $O(n^3)$ action executions.*

> **Proof Sketch:** [Koenig and Simmons, 1996] showed that the complexity of min-LRTA* is at most $2 \sum_{s \in S \setminus G} \sum_{a \in A(s)} [gd(succ(s,a)) + 1]$, i.e. $O(e \times d)$. Since $e \leq n^2$ and $d \leq n-1$, it follows that the complexity is at most $O(n^3)$. Reset state spaces with $O(n^2)$ actions are examples of state spaces for which min-LRTA* needs at least $O(n^3)$ action executions in the worst case in order to reach the goal state, as will be shown in the next section. ■

5.1.2 Efficient Search Algorithms

No real-time search algorithm that fits our real-time search skeleton can distinguish between actions that have not been executed, because it does not look at the successor states of its current state when choosing actions (and initially all actions have the same value). This implies the following lower bound on their complexity.

Theorem 2 *The complexity of every real-time search algorithm that fits our real-time search skeleton is at least $O(n^3)$ action executions.*

> **Proof Sketch:** Reset state spaces with $O(n^2)$ actions, see Figure 7, are examples of state spaces for which every real-time search algorithm that fits our real-time search skeleton needs at least $O(n^3)$ action executions in the worst case in order to reach the goal state. In particular, every real-time search algorithm can traverse either the state sequence that is printed by the following program in pseudo code or a super sequence thereof if ties are broken in favor of successor states with smaller numbers.[2]

```
for i := 1 to n-1
  print i
  for j := 1 to i-1
    for k := j to i
      print k
print n
```

[2]The scope of the for-statements is shown by indentation. The statements in their scope only get executed if the range of the for-variable is not empty. The step size for for-statements is either one ("to") or minus one ("downto").

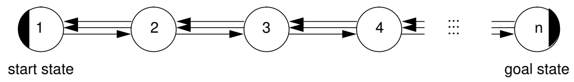

Figure 8: A quicksand state space

In this case, all real-time search algorithms execute at least $1/6 \times n^3 - 1/6 \times n$ actions before they reach the goal state (for $n \geq 1$). ■

Thus, no real-time search algorithm can have a complexity smaller than $O(n^3)$ action executions and beat min-LRTA*, that we have shown to have a tight complexity of $O(n^3)$ action executions. (We do not pursue the question here whether there exist search algorithms whose performance dominates the one of min-LRTA*, see [Koenig and Simmons, 1996] for the answer.)

5.1.3 Inefficient Search Algorithms

In this section, we analyze the complexity of inefficient real-time search algorithms. Note that "the worst real-time search algorithm" does not exist, since one can construct algorithms that perform arbitrarily badly, even if they fit our real-time search skeleton. This problem should be addressed by performing a complexity analysis over a suitably defined class of "reasonable" real-time search algorithms. In this report, however, we are content with analyzing examples of inefficient real-time search algorithms.

Particularly bad search algorithms are ones that do not remember where they have already searched. Random walks are examples of such search algorithms. We show that their deterministic counterpart is a real-time search algorithm called "edge counting," and demonstrate that both algorithms are intractable in reset state spaces. Another class of state spaces in which their performance degrades completely are "quicksand" state spaces. In every state of a quicksand state space (except for the boundary), there are more actions that move the agent one action execution away from the goal state ("a bit into the quicksand") than move it one action execution towards it ("a bit out of the quicksand").[3] An example of a quicksand state space is shown in Figure 8. Quicksand state spaces differ from reset state spaces in the effort that is necessary to recover from mistakes: all actions in quicksand state spaces have local effects only (it is possible to recover in only one step), whereas the reset actions in reset state spaces do not have local effects.

[3]Quicksand state spaces are also crude, discrete models of navigation tasks that require an agent to travel some distance on a conveyor belt that moves away from the goal location. If the agent can always select its travel speed from the interval $[-v_{max}, v_{max}]$, then the majority of its actions increase its goal distance.

5.1.3.1 Random Walks Random walks choose randomly from the available actions.

Random Walks	
action selection step (line 3)	$a :=$ pick an action from $A(s)$ with uniform probability
value update step (line 4)	(empty)

Random walks have lookahead zero, do not store any information at all, and thus cannot remember where they have already searched unsuccessfully. As a consequence, the number of action executions they need in order to reach a goal state can exceed every given bound with positive probability, implying an infinite (worst-case) complexity. However, random walks find a goal state with probability one and finite average-case complexity, although the expected number of action executions can scale exponentially in the size of the state space.

Theorem 3 *Random walks have infinite (worst-case) complexity. Their average-case complexity is finite, but can be exponential in n.*

Proof Sketch: It is easy to show that random walks have infinite (worst-case) complexity, but finite average-case complexity (proof trivial). Simple reset state spaces, see Figure 6, are examples of state spaces for which random walks need a number of action executions on average that is exponential in n in order to reach a goal state. This can be seen as follows: For every $s \in S$ we introduce a variable x_s that represents the average number of action executions until the random walk reaches a goal state if it starts in state s. These values can be calculated by solving the following set of linear equations.

$$\begin{aligned} x_1 &= 1 + x_2 \\ x_s &= 1 + 0.5x_1 + 0.5x_{s+1} \qquad \text{for all } s \in \{2, 3, \ldots, n-1\} \\ x_n &= 0 \end{aligned}$$

The result is that random walks execute on average $x_1 = 3 \times 2^{n-2} - 2$ actions before they reach the goal state (for $n \geq 2$). – Quicksand state spaces, see Figure 8, are another kind of state spaces for which random walks need a number of action executions on average that is exponential in n in order to reach a goal state. We proceed as we did for reset state spaces: we solve a set of linear equations to calculate the average-case complexity of random walks. For every $s \in S$, we introduce a variable x_s that represents the average number of action executions until the random walk reaches a goal state if it starts in state s.

$$\begin{aligned} x_1 &= 1 + x_2 \\ x_s &= 1 + 2/3 \times x_{s-1} + 1/3 \times x_{s+1} \qquad \text{for all } s \in \{2, 3, \ldots, n-1\} \\ x_n &= 0 \end{aligned}$$

The result is that random walks execute on average $x_1 = 2^{n+1} - 3n - 1$ actions before they reach the goal state (for $n \geq 1$). ■

5.1.3.2 Edge Counting We can easily derive a real-time search algorithm that shares many properties with random walks, but has finite complexity – basically, by "removing the randomness" from random walks.

Edge Counting	
action selection step (line 3)	$a :=$ one-of $\arg\min_{a' \in A(s)} V(s, a')$
value update step (line 4)	$V(s, a) := 1 + V(s, a)$

Random walks execute all actions in a state equally often in the long run. The action selection step of edge counting always chooses the action for execution that has been executed the least number of times. This achieves the same result as random walks, but in a deterministic way. One particular tie breaking rule, for example, is to execute all actions in turn. Shannon used this algorithm as early as in the late 1940's to implement an exploration behavior for an electronic mouse that searched a maze, see [Sutherland, 1969]. To the best of our knowledge, however, its relationship to random walks has never been pointed out nor has its complexity been analyzed.

Edge counting shares many properties with random walks. In particular, edge counting tends to have the properties in the worst case that random walks have on average. For example, the (worst-case) complexity of edge counting over all state spaces (in big-O notation) equals the average-case complexity of random walks. In particular, edge counting always reaches a goal state with a finite number of action executions, but its complexity can be exponential in the size of the state space. (We do expect some improvement in average-case performance when switching from random walks to edge counting, since edge counting remembers something about where it has already searched. The improvement, however, can just be a constant factor. Section 6 contains experimental results.)

Theorem 4 *The complexity of edge counting is finite, but at least exponential in n.*

Proof Sketch: The argument that edge counting reaches a goal state eventually is by contradiction: If edge counting did not reach a goal state eventually, it would execute actions forever. In this case, there is a time t from which on edge counting only executes those actions that it executes infinitely often. Eventually, the values of all of these actions exceed every bound, since – every time an action is executed – its value is incremented by one. In particular, the values exceed the value of an action that edge counting considers infinitely often for execution, but never executes after time t. Such an action exists, since one can reach a goal state from every state. Then, however, edge counting is forced to execute this action after time t, which is a contradiction.

Simple reset state spaces, see Figure 6, are examples of state spaces for which edge counting needs a number of action executions in the worst case that is at least exponential in n. In particular, edge counting traverses the state sequence printed by `f(n)` if ties are broken in favor of successor states with smaller numbers.

```
proc f(i) =
  if i = 2 then
    print 1
  else
    f(i-1)
    f(i-1)
  print i
```

In this case, edge counting executes $3 \times 2^{n-2} - 2$ actions before it reaches the goal state (for $n \geq 2$). Similarly, the state space in Figure 4 is a reset state space for which edge counting executes $3 \times 2^{1/3 \times n - 1/3} - 4$ actions before it reaches the goal state (for $n \geq 4$ with $n \bmod 3 = 1$) if ties are broken in favor of actions that lead "upward."

Quicksand state spaces, see Figure 8, are another kind of state spaces for which edge counting needs a number of action executions in the worst case that is at least exponential in n. In particular, edge counting traverses the state sequence that is printed by the following program in pseudo code if ties are broken in favor of successor states with smaller numbers.

```
print 1
print 2
for i := 3 to n
  print i-2
  f(i-1)
  print i-2
  f(i-1)
  print i
```

where

```
proc f(i) =
  if i = 2 then
    print 2
  else
    print i-2
    f(i-1)
    print i-2
    f(i-1)
    print i
```

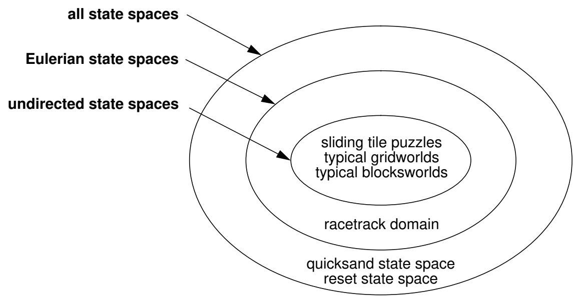

Figure 9: Subsets of state spaces (including example domains)

In this case, edge counting executes $2^{n+1} - 3n - 1$ actions before it reaches the goal state (for $n \geq 1$). ■

To summarize, edge counting needs a number of action executions that is, in the worst case, exponential in n for both (simple) reset state spaces (Figure 6) and quicksand state spaces (Figure 8). These are Type 2 state spaces since, by Theorem 1, min-LRTA* needs only a polynomial number of action executions in these state spaces.

5.2 Undirected and Eulerian State Spaces

In this section, we consider the complexity of real-time search algorithms in a subset of state spaces, namely state spaces with $d < \infty$ that are either undirected or Eulerian (we do not assume that the real-time search algorithms know that the state spaces have one of these properties), and show that they are all of Type 1b.[4]

Definition 1 *A state space is* **Eulerian** *iff* $|A(s)| = |\{(s', a') : s' \in S \wedge a' \in A(s') \wedge succ(s', a') = s\}|$ *for all* $s \in S$*, i.e. there are as many actions that leave a state as there are actions that enter the (same) state.*

Since an undirected edge is equivalent to one incoming edge and one outgoing edge, all undirected state spaces are Eulerian. Many state spaces of typical AI search domains are undirected (and thus Eulerian), see Figure 9. Examples include sliding tile puzzles and typical gridworlds, i.e. the state spaces that are commonly used as testbeds for LRTA*, edge counting, and other real-time search algorithms. Gridworlds, for example, are very popular abstractions of robot navigation domains. They discretize 2-d space into square cells, see Figure 10. A robot can move from any square to each of its

[4] Eulerian state spaces correspond to directed Euler(ian) graphs as defined by the Swiss mathematician Leonhard Euler when he considered whether the seven Königsberg bridges could be traversed without recrossing any of them [Newman, 1953].

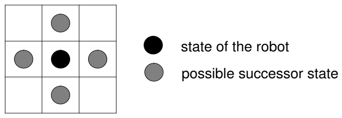

Figure 10: Gridworld

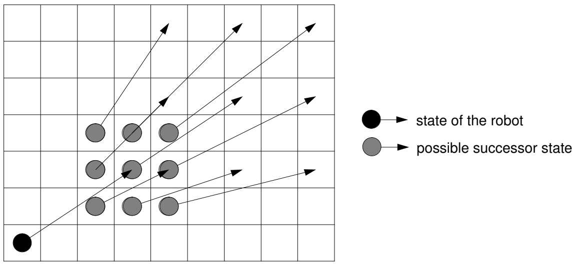

Figure 11: Racetrack domain

four neighboring squares as long as it stays on the grid and the target square does not contain an obstacle.

There also exist state spaces that are Eulerian, but not undirected, for example the state spaces of racetrack domains [Gardner, 1973]. They correspond to gridworlds, but a state of the state space is not only characterized by the x and y coordinates of the square that the robot currently occupies. Instead, it is described by two pairs of integers: the square that the robot occupies, and its speed in both the x and y direction. Actions correspond to adjusting both the x and y speed components by -1, 0, or 1. Given an action (speed change), the successor state is determined by computing the new speed components (one can impose a limit that the absolute speeds are not allowed to exceed) and then determining the new location of the robot by adding each speed component to its corresponding location component. An example is shown in Figure 11. Racetrack domains are robot navigation domains that are more realistic than gridworlds – for instance, they model acceleration and take into account that the turn radius of the robot gets larger at higher speeds. The state spaces of racetrack domains are Eulerian except around obstacles or at boundaries. In particular, the state spaces of obstacle free racetrack domains on a torus are truly Eulerian, but not undirected. Race track domains have been used as testbeds for real-time search algorithms by [Barto *et al.*, 1995].

The complexity of real-time search algorithms in a subset of state spaces can potentially be smaller than their complexity in general. This is the case if all state spaces on which the real-time search algorithms do not perform well do not belong to the subset. Thus, Eulerian state spaces could be easier to search than state spaces in general, and

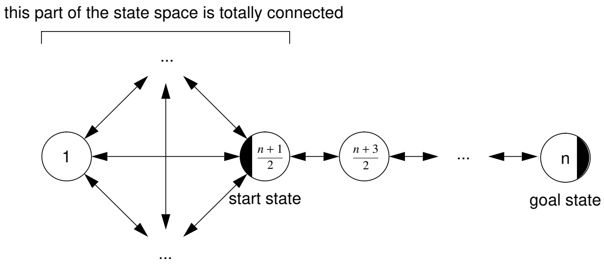

Figure 12: An undirected state space

undirected state spaces could be even easier to search. We now show that Eulerian state spaces are indeed easier to search with the studied real-time search algorithms than state spaces in general, but undirected state spaces do not simplify the search any further. Since there are some search problems that are of Type 2, they must be non-Eulerian. Examples include reset state spaces and quicksand state spaces.

5.2.1 LRTA*-Type Real-Time Search

The complexity of min-LRTA* does not decrease in undirected or Eulerian state spaces.

Theorem 5 *Min-LRTA* has a tight complexity of $O(n^3)$ action executions in undirected or Eulerian state spaces.*

Proof Sketch: Theorem 1 implies that the complexity of min-LRTA* is at most $O(n^3)$ action executions. Figure 12 shows an example of an undirected (and thus Eulerian) state space for which min-LRTA* needs at least $O(n^3)$ action executions in the worst case in order to reach the goal state. In particular, it traverses the state sequence that is printed by the following program in pseudo code.

```
for i := n-1 downto (3n+1)/4
  print (n+1)/2
  for j := 1 to (n-1)/2
    for k := j+1 to (n+1)/2
      print j
      print k
  for j := (n+3)/2 to i-1
    print j
  for j := i downto (n+3)/2
    print j
```

```
print (n+1)/2
for j := 1 to (n-1)/2
  for k := j+1 to (n+1)/2
    print j
    print k
for j := (n+3)/2 to n
  print j
```

In this case, min-LRTA* executes $1/16 \times n^3 + 3/8 \times n^2 - 3/16 \times n - 1/4$ actions before it reaches the goal state (for $n \geq 1$ with $n \bmod 4 = 1$). ∎

5.2.2 Efficient Search Algorithms

For Eulerian state spaces, real-time search algorithms exist that have a lower complexity than min-LRTA*. One example, called BETA[5] ("Building a Eulerian Tour" Algorithm), informally acts as follows:

> Take unexplored edges whenever possible. If stuck [i.e. if all actions in the current state have already been executed at least once], retrace the closed walk of unexplored edges just completed, stopping at nodes that have unexplored edges, and apply this algorithm recursively from each such node.

This algorithm is similar to depth-first search, with the following difference: Since chronological backtracking is not always possible in directed graphs, BETA repeats its first actions when it gets stuck instead of backtracking its latest actions.

BETA fits our real-time search skeleton, as we show in the following. We encode the values $V(s, a)$ as triples of integers. (Such a triple is then encoded as one integer, which we don't show here.) The first component (the "cycle number") has index one and corresponds to the level of recursion of the recursive version of BETA as stated above. The second component has index two and counts the number of times action a has already been executed, and the third component remembers when action a was executed first (using a counter that is incremented after every action execution). The variable *memory* is also treated as a triple: its first two components remember the first two components of the previously executed action and its third component is the counter. All values are initialized to $(0, 0, 0)$ instead of 0. (There are more concise ways of representing the necessary information, but they tend to be more difficult to comprehend.)

[5]The exact origin of the algorithm is unclear. [Deng and Papadimitriou, 1990] and [Korach *et al.*, 1990] stated it explicitly as a search algorithm, but it has been used much earlier as part of proofs about properties of Eulerian graphs [Hierholzer, 1873].

BETA	
action selection step (line 3)	$a :=$ one-of $\arg\min_{a' \in X} V(s,a')[3]$ where $X = \arg\max_{a' \in Y} V(s,a')[1]$ and $Y = \arg\min_{a' \in A(s)} V(s,a')[2]$

value update step (line 4)	if V(s,a)[2] = 0 then
	  V(s,a)[3] := memory[3]+1
	  if memory[2] = 1 then
	    V(s,a)[1] := memory[1]
	  else then
	    V(s,a)[1] := memory[1]+1
	V(s,a)[2] := V(s,a)[2] + 1
	memory[1] := V(s,a)[1]
	memory[2] := V(s,a)[2]
	memory[3] := memory[3] + 1

The action selection step always chooses an action that has never been executed before. (We do not care how ties are resolved.) If no such action exists in the current state, it considers all actions that have been executed exactly once and selects the action that belongs to the latest cycle. Such an action always exists. If there is more than one such action, ties are broken in favor of the action whose first execution preceded the executions of the other actions. When an action is executed for the first time, the value update step remembers when it was executed and decides which cycle number it should get. If the action executed previously was executed for the first time too, then the new action inherits its cycle number, otherwise a new cycle starts and the cycle number of the new action is one larger than the cycle number of the previously executed action. For every action, the value update step also increments the number of times it has been executed. Finally, the value update step remembers the first two components of the current action (so that it has them available after the action has been executed) and increments the system clock.

BETA always reaches a goal state with a finite number of action executions and, moreover, executes every action at most twice. Furthermore, every real-time search algorithm that fits our real-time search skeleton needs asymptotically at least this many action executions [Deng and Papadimitriou, 1990]. These two statements remain true if only undirected state spaces are considered [Koenig and Smirnov, 1996]. The following theorem follows.

Theorem 6 *The complexity of every real-time search algorithm that fits our real-time search skeleton is at least $O(n^2)$ action executions in undirected or Eulerian state spaces, and BETA has a tight complexity of $O(n^2)$ action executions in these state spaces.*

Proof Sketch: According to [Deng and Papadimitriou, 1990], BETA executes every action at most twice in Eulerian state spaces. Since $e \leq n^2$, its complexity is at most $O(n^2)$ in undirected or Eulerian state spaces.

On the other hand, it is easy to construct undirected (and thus Eulerian) state spaces with $O(n^2)$ actions most of which BETA or any other real-time search algorithm that fits our framework has to traverse at least once in the worst-case before it reaches a goal state. ■

5.2.3 Inefficient Search Algorithms

In the following, we analyze the real-time search algorithms again that we have shown to be very inefficient in general, namely random walks and edge counting.

5.2.3.1 Random Walks The (worst-case) complexity of random walks remains infinite in undirected or Eulerian state spaces, but their average-case complexity decreases dramatically from being exponential in n to being a small polynomial in n.

Theorem 7 *Random walks have infinite (worst-case) complexity in undirected or Eulerian state spaces. Their average-case complexity is tight at $O(n^3)$ action executions in these state spaces.*

Proof Sketch: It is easy to show that random walks have infinite (worst-case) complexity, but finite average-case complexity (proof trivial). According to [Aleliunas *et al.*, 1979], a random walk needs at most e action executions on average in Eulerian state spaces in order to reach a specified successor state of the state it started in. Now consider a shortest path from the start state to the closest goal state. Since its length is at most d, a random walk needs at most $e \times d$ action executions on average to reach the goal state. Since $e \leq n^2$ and $d \leq n - 1$, the average-case complexity of random walks is at most $O(n^3)$ action executions.

Figure 12 shows an example of an undirected (and thus Eulerian) state space for which random walks need $O(n^3)$ action executions on average in order to reach the goal state. As in the proof of Theorem 3, we can solve a system of linear equations to calculate the average-case complexity of random walks in this state space. The result is that random walks execute $1/8 \times n^3 + 1/8 \times n^2 - 5/8 \times n + 3/8$ actions on average before they reach the goal state (for odd $n \geq 1$). ■

5.2.3.2 Edge Counting Since the average-case complexity of random walks decreases in undirected or Eulerian state spaces from being exponential in n to being a small polynomial in n, so does the complexity of edge counting.

Theorem 8 *Edge counting has a tight complexity of $O(n^3)$ action executions in undirected or Eulerian state spaces.*

Proof Sketch: According to Theorem 9 in the appendix, the complexity of edge counting is at most $e \times gd(s_{start}) - gd(s_{start})^2$ action executions in undirected or Eulerian state spaces. Since $e \leq n^2$ and $gd(s_{start}) \leq d \leq n-1$, it follows that its complexity is at most $O(n^3)$.

Figure 12 shows an example of an undirected (and thus Eulerian) state space for which edge counting needs at least $O(n^3)$ action executions in the worst case in order to reach the goal state. In particular, it traverses the state sequence that is printed by the following program in pseudo code if ties are broken in favor of successor states with smaller numbers.

```
for i := (n+3)/2 to n
  for j := i-2 downto (n+1)/2
    print j
  for j := 1 to (n-1)/2
    for k := j+1 to (n+1)/2
      print j
      print k
  for j := (n+3)/2 to i
    print j
```

In this case, edge counting executes $e \times gd(s_{start}) - gd(s_{start})^2 = 1/8 \times n^3 + 1/8 \times n^2 - 5/8 \times n + 3/8$ actions before it reaches the goal state (for odd $n \geq 1$). ■

6 Average-Case Results

So far, we have only been concerned with the (worst-case) complexity of real-time search algorithms. However, their average-case complexity is equally important for practical purposes, and worst-case performance guarantees do not necessarily reflect average-case performance. To show that the average-case performance follows a similar trend, we present a simple case study that demonstrates that Eulerian state spaces are not only simpler to search than reset state spaces in the worst case, but also on average.

Figures 13 and 14 show how many actions four real-time search algorithms execute in the two blocksworld domains from Figures 3 (Domain 1) and 4 (Domain 2). We compare min-LRTA*, random walks, edge counting, and – for the Eulerian state space of Domain 1 – also BETA. All graphs are scaled in the same proportions. Their horizontal axes show the size of the state space (measured by the number of blocks) and their vertical axes the number of action executions until a goal state was reached from the start state, averaged over 5000 runs with randomly broken ties. Note that all real-time search algorithms were uninformed – in particular, they initially had no knowledge that putting blocks on the stack is the best way to achieve the goal state.

The experiments show that the relationship of the average-case performances are similar to those in the worst case. Every algorithm does better in Domain 1 than in Domain 2.

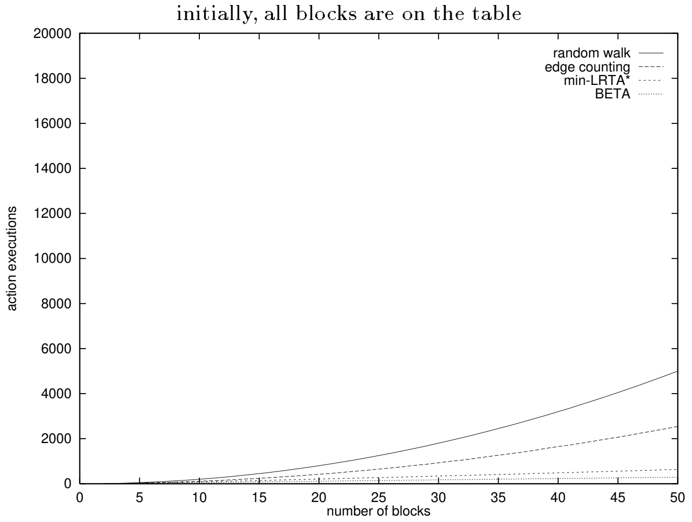

Figure 13: Performance results for Domain 1

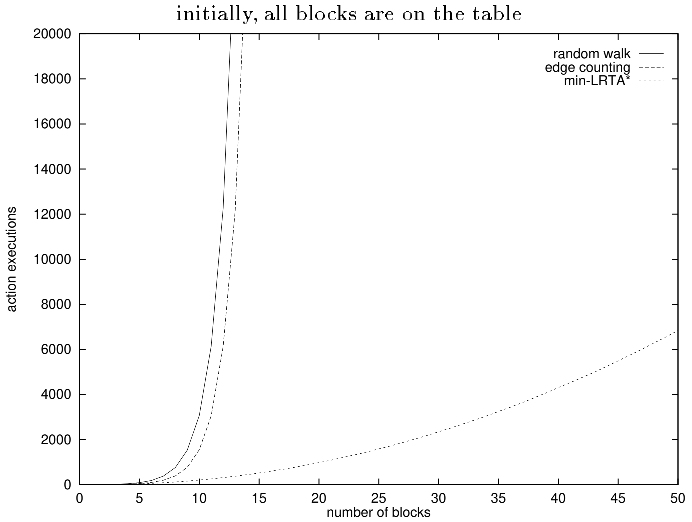

Figure 14: Performance results for Domain 2

Random walks perform worst in both state spaces. This is to be expected since they do not remember any information. However, edge counting performs almost as poorly as random walks in Domain 2, and both algorithms quickly become intractable. With 50 blocks, for example, a random walk needs about 3.4×10^{15} action executions on average in order to reach the goal state and performs about 500,000,000,000 (500 billion) times worse than min-LRTA*, that needs only 6838.3 action executions on

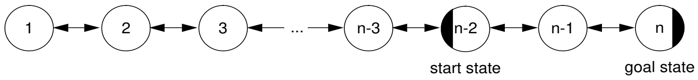

Figure 15: A linear state space

average. On the other hand, all algorithms do quite well in Domain 1. Even the ones that perform poorly in Domain 2 perform almost as well as min-LRTA*, the real-time search algorithm that performs well in both state spaces. With 50 blocks, for example, min-LRTA* performs 2.2 times worse than BETA (that needs 292.0 action executions on average), edge counting performs 8.7 times worse, and even random walks perform only 17.1 times worse. Thus, the interval spanned by the average-case complexity of efficient and inefficient real-time search algorithms is much smaller in Domain 1 than in Domain 2. This difference is to be expected, since Domain 1 is Eulerian (and thus of Type 1b), whereas Domain 2 resembles a simple reset state space of Type 2.

7 Another Search Problem

Our study does not imply that min-LRTA* has a smaller complexity than edge counting in every state space of a given size. A simple counterexample is given in Figure 15. (Another example is the state space shown in Figure 12 with start state $n-2$.) Min-LRTA* can traverse the state sequence printed by the following program in pseudo code if ties are broken in favor of successor states with smaller numbers on the (undirected) line except for the first action execution in which the tie is broken in the opposite way.

```
print n-2
for i := n-1 to 1
  print i
for i := n-2 downto 2
  for j := 2 to i
    print j
  for j := i-1 downto 1
    print j
for i := 2 to n
  print i
```

In this case, min-LRTA* executes $n^2 - 3n + 4$ actions before it reaches the goal state (for $n \geq 3$). On the other hand, we have shown that edge counting is guaranteed not to need more than $e \times gd(s_{start}) - gd(s_{start})^2 = 4n - 8$ action executions in order to reach a goal state (this bound turns out to be tight for this particular state space if ties are broken in favor of successor states with smaller numbers). Since $n^2 - 3n + 4 > 4n - 8$ for $n > 4$, the complexity of edge counting for this particular search problem is guaranteed

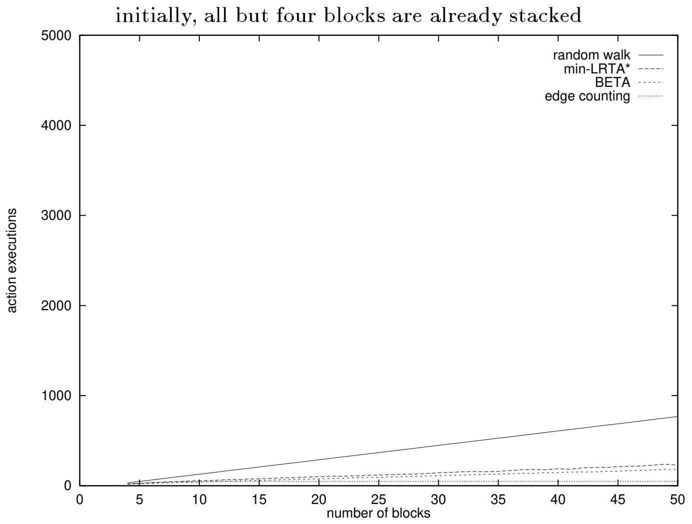

Figure 16: Performance results for Domain 1

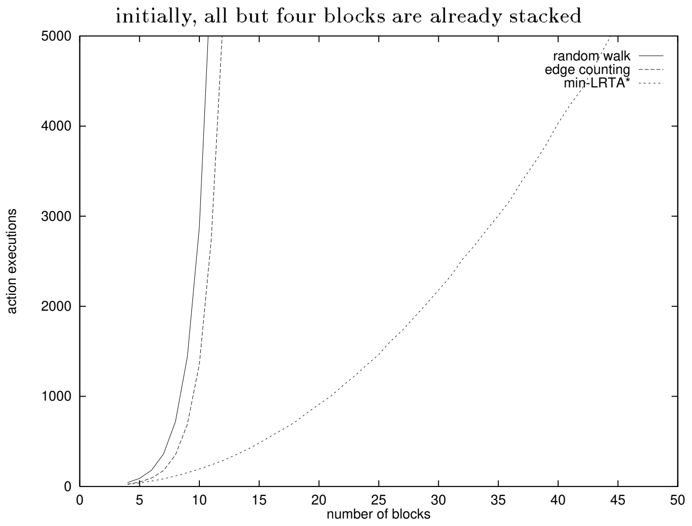

Figure 17: Performance results for Domain 2

to be smaller than that of min-LRTA* for $n > 4$. Experiments show that the same relationship holds, even more pronounced, in the average case.

Consider again the two blocksworld domains from Figures 3 (Domain 1) and 4 (Domain 2). If we change the start state in both domains so that all but four blocks are already stacked initially, then both domains become easier to solve. However, Figure 17 shows that the performance relationships of the real-time search algorithms

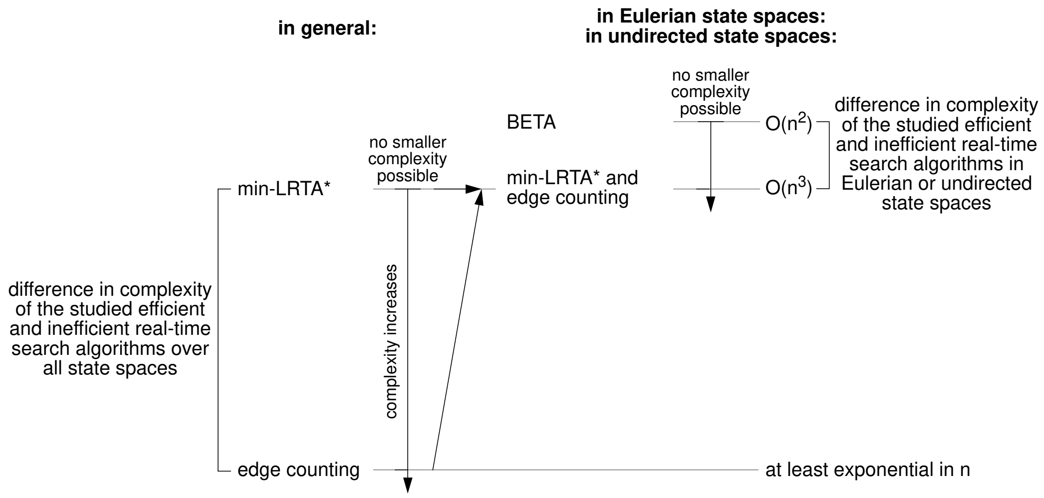

Figure 18: Worst-case performance of real-time search algorithms

studied in Section 6 remain similar in Domain 2 (min-LRTA*, for example, now needs 6414.1 action executions on average in order to reach the goal state). Figure 16, on the other hand, shows that the performance relationships in Domain 1 change dramatically. (Figures 16 and 17 are scaled differently than Figures 13 and 14.) With 50 blocks, for example, min-LRTA* now performs 1.3 times worse than BETA and random walks perform 4.2 times worse, but edge counting performs 3.8 times *better* than BETA. Thus, for this particular search problem in Domain 1 (a Eulerian state space), a real-time search algorithm that can be intractable (edge counting) outperforms a real-time search algorithm that is always efficient (min-LRTA*).

8 Summary of the Results

When comparing the complexity of min-LRTA* to the complexity of efficient and inefficient real-time search algorithms, we derived the following results. In general, no real-time search algorithm can beat the complexity of min-LRTA*, which is a small polynomial in n. In contrast, the deterministic real-time search algorithm (edge counting) that we derived from random walks has a complexity that is at least exponential in n.

The picture changes in Eulerian state spaces. The complexity of edge counting decreases dramatically and equals the complexity of min-LRTA*, which remains unchanged (it even beats min-LRTA* in certain specific state spaces). In addition, there exists a dedicated real-time search algorithm for Eulerian state spaces (BETA) that has a smaller complexity. All complexities remain the same in undirected state spaces, a subset of Eulerian state spaces.

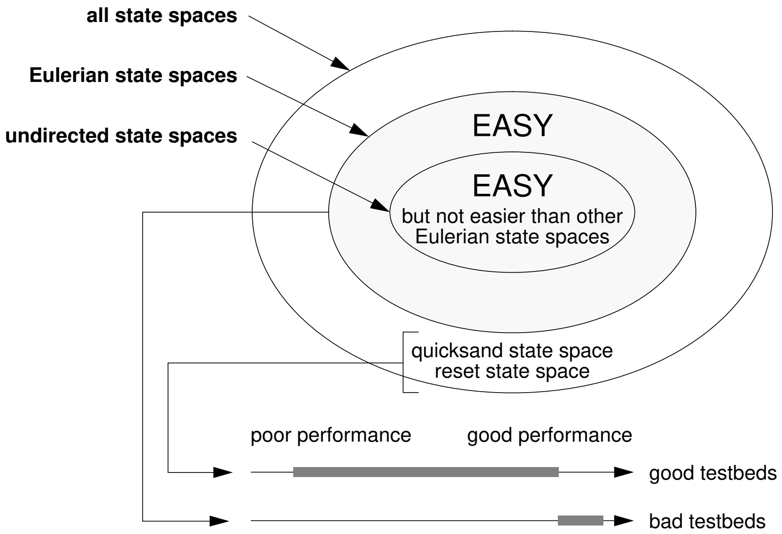

Figure 19: Good and bad testbeds (2)

To summarize, while min-LRTA* does rather well in general when being compared to the other real-time search algorithms, its advantage decreases in undirected or Eulerian state spaces, in which the complexities of real-time search algorithms span a much smaller interval than in state spaces in general, see Figure 18. In particular, the complexities of all analyzed real-time search algorithms are small polynomials in n for undirected or Eulerian state spaces, even the ones of real-time search algorithms that can be intractable, and intractable algorithms can even outperform min-LRTA*, a real-time search algorithm that is always efficient. Thus, undirected and Eulerian state spaces are easier to search with the studied real-time search algorithms than some non-Eulerian state spaces such as, for example, reset state spaces and quicksand state spaces, see Figure 19. Consequently, neither undirected nor Eulerian state spaces are ideal testbeds for comparing (some) real-time search algorithms across a wide range of state spaces.

While it is important to understand what our study implies, it is equally important to understand what its limitations are. For example, one cannot compare real-time search algorithms with each other or with off-line (traditional) search algorithms solely on the basis of our study. This is due to the fact that we limited our investigation to uninformed real-time search algorithms with minimal lookahead. It would be unfair to compare real-time search algorithms with each other solely on the basis of our study, since some algorithms are better than others in incorporating initial knowledge of the state space (for example, in the form of heuristic values for the goal distances) or allowing for larger lookaheads. LRTA*, for example, allows one easily to do both and, in addition, determines shortest paths if it is run repeatedly. It would be equally unfair to compare real-time search algorithms with off-line search algorithms on the basis of our study, since uninformed real-time search algorithms with minimal lookahead are the most inefficient real-time search algorithms possible. In the next section, we relax some of our assumptions when we discuss real-time search algorithms with larger lookaheads.

9 Real-Time Search with Larger Lookaheads

Although we have limited ourselves in this report to comparing real-time search algorithms with minimal lookahead, we would like to point out briefly that some of our results also transfer to real-time search algorithms with larger lookaheads. In the following, we discuss both node counting, a variant of edge counting, and the original 1-step LRTA* algorithm, a variant of min-LRTA*. Both algorithms have been used in the literature and have a larger lookahead than their relatives.

Node counting differs from edge counting in that it looks at the successor states of its current state when choosing actions.

Node Counting	
action selection step (line 3)	$a :=$ one-of $\arg\min_{a' \in A(s)} \sum_{a'' \in A(succ(s,a'))} V(succ(s,a'),a'')$
value update step (line 4)	$V(s,a) := 1 + V(s,a)$

The action selection step always executes the action that leads to the successor state that has been visited the least number of times. In an actual implementation, one would maintain only one value $V(s)$ for each state s with $V(s) = \sum_{a \in A(s)} V(s,a)$. In this case, initially, $V(s) = 0$ for all $s \in S$.

Node Counting	
action selection step (line 3)	$a :=$ one-of $\arg\min_{a' \in A(s)} V(succ(s,a'))$
value update step (line 4)	$V(s) := 1 + V(s)$

We compare node counting to Korf's original LRTA* algorithm with lookahead one (1-step LRTA*), a variant of min-LRTA*. 1-Step LRTA* is similar to node counting in that it looks at the successor states of its current state when choosing actions, but it has a different value update step. Initially, $V(s) = 0$ for all $s \in S$ and $a \in A(s)$.

1-Step LRTA*	
action selection step (line 3)	$a :=$ one-of $\arg\min_{a' \in A(s)} V(succ(s,a'))$
value update step (line 4)	$V(s) := 1 + V(succ(s,a))$

The state values $V(s)$ are lower bounds on $gd(s)$. Korf showed that 1-step LRTA* always reaches a goal state with a finite number of action executions. According to [Koenig and Simmons, 1995], its complexity is tight at $n^2 - n$ and remains tight at $O(n^2)$ for undirected or Eulerian state spaces.

Although we do not know of any complexity analysis for node counting in undirected or Eulerian state spaces, it has been applied in the literature to state spaces with these

1	2	3
8		4
7	6	5

Figure 20: Eight puzzle with the American goal state

properties. For example, variations of node counting have been used independently by [Pirzadeh and Snyder, 1990] and [Thrun, 1992] to explore unknown gridworlds (either on their own or to accelerate reinforcement-learning methods), in both cases with great success. Our experiments confirm these results.

- In one experiment, we compared 1-step LRTA* and node counting on an empty gridworld of size 50 times 50. We counted the number of action executions until the algorithms reached the upper left square, averaged over 25000 runs with randomly broken ties. The same 25000 randomly selected start states were used in both cases. Node counting needed, on average, 2874 action executions to reach the goal state, compared to 2830 action executions needed by 1-step LRTA*. Out of the 25000 runs, node counting outperformed 1-step LRTA* 12345 times, was beaten 12621 times, and tied 34 times.

- We performed the same experiment on the eight puzzle with the American goal state (a state space with 181440 states), see Figure 20. Node counting needed, on average, 85579 action executions to reach the goal state, compared to 85746 action executions needed by 1-step LRTA*. Out of the 25000 runs, node counting outperformed 1-step LRTA* 12512 times and was beaten 12488 times.

In both experiments, the performance differences between node counting and 1-step LRTA* prove to be statistically insignificant for any reasonable level of significance (no matter whether one uses a sign test or a t test). However, we can show that node counting is similar to edge counting in that there exist state spaces for which its complexity is at least exponential in n. In particular, we engineered our blocksworld domains from Figures 3 and 4 so that node counting and edge counting behave identically. This is the only reason why we used two-stage operators in the blocksworld domain instead of one-stage operators. The appearance of the intermediate "pickup" operators makes it so that a 1-step lookahead is insufficient to avoid reset traps. In particular, all theorems for edge counting in Eulerian state spaces also apply to node counting in state spaces that have been derived from Eulerian state spaces by replacing every directed edge with two directed edges that are connected with an intermediate vertex. Thus, for these search problems node counting has the same worst-case and average-case complexity than edge counting. For example, both algorithms are efficient in Domain 1 and exponential in Domain 2 if ties are broken appropriately. This means that the variant of the simple reset state space that is shown in Figure 4 is an example of a state space for which node counting needs a number of action executions in order to reach the goal state that is exponential in n if ties are broken in favor of actions that lead "upward." Since the complexity of 1-step LRTA* is always at most $O(n^3)$, variants of reset state spaces and quicksand state spaces again separate real-time search algorithms that are always efficient (here: 1-step LRTA*) from ones that can be intractable (here: node counting).

To summarize, 1-step LRTA* and node counting are almost equally efficient on both gridworlds and sliding tile puzzles, but reset and quicksand state spaces are able to differentiate between them. Similar reset state spaces and quicksand state spaces can also be constructed for real-time search algorithms with even larger lookaheads.

10 Conclusion

This report investigated whether there are properties of state spaces that make them bad testbeds for real-time search algorithms, in an attempt to separate the inherent complexity of a given search problem from the performance of individual real-time search algorithms. We made a first step in this direction by comparing several uninformed real-time search algorithms with minimal lookahead that solve suboptimal search problems – all algorithms had previously been used by different researchers in different contexts. More precisely, we compared variants of LRTA* to both efficient real-time search algorithms such as BETA, and – equally importantly – inefficient real-time search algorithms such as edge counting (a deterministic algorithm that we showed to share many properties with random walks). Our analysis demonstrated that one can learn not only from comparing search algorithms to the best known ones – as it is usually done – but also from comparing them to inefficient ones.

We demonstrated, both theoretically and experimentally, that the performance characteristics of the studied real-time search algorithms can differ significantly in Eulerian and non-Eulerian state spaces (real-time search algorithms differ in this respect from traditional search algorithms such as the A* algorithm). We have shown that real-time search algorithms that can be intractable in non-Eulerian state spaces (such as edge counting) have a small complexity in Eulerian state spaces (a superset of undirected state spaces) and can even outperform those algorithm that are tractable in all state spaces (such as min-LRTA*, a variant of LRTA*). A more detailed summary of the results was given in Section 8.

Our results help to explain why the reported performance of some real-time search algorithms has been so good: they tended to be tested in Eulerian state spaces such as traditional AI search domains (sliding tile puzzles and gridworlds, for example). Many state spaces, however, are not undirected or Eulerian (examples include most state spaces of applications in the field of control theory). One way to avoid uncritical generalizations of performance figures for real-time search algorithms by non-experts is to report experimental results not only for Eulerian state spaces, but also for non-

Eulerian state spaces. In particular, one has to use non-Eulerian state spaces to show the superiority of a particular real-time search algorithm across a wide range of domains. To this end, we presented two classes of state spaces ("reset state spaces" and "quicksand state spaces") that are able to separate some intractable real-time search algorithms with minimal lookahead from tractable ones and, thus, do not suffer from (all of) the problems of the standard test domains. Furthermore, minor variations of these domains can also separate real-time search algorithms with larger lookaheads. Thus, while we cannot expect any single domain to be sufficient for comparing real-time search algorithms[6], we do suggest that variations of reset state spaces and quicksand state spaces be included in test suites for real-time search algorithms.

To summarize, our study provides a first step in the direction of understanding how properties of state spaces influence the performance of real-time search algorithms. In this report, we reported results for one particular property: being Eulerian. Our current work concentrates on studying additional properties that occur in more realistic applications, such as real-time control.

Acknowledgements

Thanks to Lonnie Chrisman, Robert Holte, Andrew Moore, and Jiri Sgall for helpful discussions.

References

(Aleliunas *et al.*, 1979) Aleliunas, R.; Karp, R.M.; Lipton, R.J.; Lovász, L.; and Rackoff, C. 1979. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the Symposium on Foundations of Computer Science*. 218–223.

(Barto *et al.*, 1995) Barto, A.G.; Bradtke, S.J.; and Singh, S.P. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 73(1):81–138.

(Deng and Papadimitriou, 1990) Deng, X. and Papadimitriou, C.H. 1990. Exploring an unknown graph. In *Proceedings of the Symposium on Foundations of Computer Science*. 355–361.

(Gardner, 1973) Gardner, M. 1973. Mathematical games. *Scientific American* 228(1):108–115.

(Hanks *et al.*, 1993) Hanks, S.; Pollack, M.E.; and Cohen, P.R. 1993. Benchmarks, test beds, controlled experimentation, and the design of agent architectures. *AI Magazine* 14(4):17–42.

[6]In "quicksand state spaces," for example, all actions have only local effects and some inefficient real-time search algorithms might be able to perform efficiently in them.

(Hierholzer, 1873) Hierholzer, C. 1873. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen* 6:30–32.

(Ishida and Korf, 1991) Ishida, T. and Korf, R.E. 1991. Moving target search. In *Proceedings of the IJCAI*. 204–210.

(Ishida, 1992) Ishida, T. 1992. Moving target search with intelligence. In *Proceedings of the AAAI*. 525–532.

(Ishida, 1995) Ishida, T. 1995. Two is not always better than one: Experiences in real-time bidirectional search. In *Proceedings of the International Conference on Multi-Agent Systems*. 185–192.

(Knight, 1993) Knight, K. 1993. Are many reactive agents better than a few deliberative ones? In *Proceedings of the IJCAI*. 432–437.

(Koenig and Simmons, 1992) Koenig, S. and Simmons, R.G. 1992. Complexity analysis of real-time reinforcement learning applied to finding shortest paths in deterministic domains. Technical Report CMU–CS–93–106, School of Computer Science, Carnegie Mellon University.

(Koenig and Simmons, 1995) Koenig, S. and Simmons, R.G. 1995. Real-time search in non-deterministic domains. In *Proceedings of the IJCAI*. 1660–1667.

(Koenig and Simmons, 1996) Koenig, S. and Simmons, R.G. 1996. The effect of representation and knowledge on goal-directed exploration with reinforcement learning algorithms. *Machine Learning Journal* 22:227–250.

(Koenig and Smirnov, 1996) Koenig, S. and Smirnov, Y. 1996. Graph learning with a nearest neighbor approach. In *Proceedings of the Conference on Computational Learning Theory*. 19–28.

(Korach *et al.*, 1990) Korach, E.; Kutten, S.; and Moran, S. 1990. A modular technique for the design of efficient distributed leader finding algorithms. *ACM Transactions on Programming Languages and Systems* 12(1):84–101.

(Korf, 1987) Korf, R.E. 1987. Real-time heuristic search: First results. In *Proceedings of the AAAI*. 133–138.

(Korf, 1988) Korf, R.E. 1988. Real-time heuristic search: New results. In *Proceedings of the AAAI*. 139–144.

(Korf, 1990) Korf, R.E. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2-3):189–211.

(Korf, 1993) Korf, R.E. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

(Matsubara and Ishida, 1994) Matsubara, S. and Ishida, T. 1994. Real-time planning by interleaving real-time search with subgoaling. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*. 122–127.

(Newman, 1953) Newman, J. 1953. Leonhard Euler and the Königsberg bridges. *Scientific American* 188(6):66–70.

(Pearl, 1985) Pearl, J. 1985. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Menlo Park, California.

(Pemberton and Korf, 1992) Pemberton, J.C. and Korf, R.E. 1992. Incremental path planning on graphs with cycles. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*. 179–188.

(Pirzadeh and Snyder, 1990) Pirzadeh, A. and Snyder, W. 1990. A unified solution to coverage and search in explored and unexplored terrains using indirect control. In *International Conference on Robotics and Automation*, volume 3. 2113–2119.

(Russell and Wefald, 1991) Russell, S. and Wefald, E. 1991. *Do the Right Thing – Studies in Limited Rationality*. The MIT Press, Cambridge, Massachusetts.

(Stentz, 1995) Stentz, A. 1995. The focussed D* algorithm for real-time replanning. In *Proceedings of the IJCAI*. 1652–1659.

(Sutherland, 1969) Sutherland, I. 1969. A method for solving arbitrary-wall mazes by computer. *IEEE Transactions on Computers* C–18(12):1092–1097.

(Thrun, 1992) Thrun, S.B. 1992. The role of exploration in learning control with neural networks. In White, D.A. and Sofge, D.A., editors 1992, *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Reinhold, Florence, Kentucky. 527–559.

A The Complexity of Edge Counting

Before we can determine the complexity of edge counting in Eulerian state spaces, we have to introduce some definitions and preliminary results. A time superscript of t in the following theorems refers to the values of the variables immediately before the (t+1)st value update step (line 4) of edge counting, e.g. $s^{(t=)0} = s_{start}$ or $V^{(t=)0}(s,a) = 0$ for all $s \in S$ and $a \in A(s)$.

Lemma 1 *For all times t, $s \in S$, and $a \in A(s)$, $V^t(s,a) \leq \min_{a' \in A(s)} V^t(s,a') + 1$.*

Thus, the values of any two actions leaving a state differ by at most one.

Proof Sketch by induction on t: The theorem holds at time $t = 0$, since $V^0(s,a) = 0$ for all $s \in S$ and $a \in A(s)$. Assume that the theorem holds at an arbitrary time t, and consider arbitrary $s \in S$ and $a \in A(s)$. The only value that changes between t and $t+1$ is $V(s^t, a^t)$. We distinguish two cases: First, $s \neq s^t$ or $a \neq a^t$. Then, $V^{t+1}(s,a) = V^t(s,a) \overset{\text{assumption}}{\leq} \min_{a' \in A(s)} V^t(s,a') + 1 \overset{\text{monotonicity}}{\leq} \min_{a' \in A(s)} V^{t+1}(s,a') + 1$. Second, $s = s^t$ and $a = a^t$. Then, $V^{t+1}(s,a) = V^t(s,a) + 1 \overset{\text{action selection}}{=} \min_{a' \in A(s)} V^t(s,a') + 1 \overset{\text{monotonicity}}{\leq} \min_{a' \in A(s)} V^{t+1}(s,a') + 1$. Put together, $V^{t+1}(s,a) \leq \min_{a' \in A(s)} V^{t+1}(s,a') + 1$ for all $s \in S$ and $a \in A(s)$, and the theorem holds at time $t+1$ as well. ∎

Lemma 2 *For all times t, $\max_{a \in A(s^t)} V^{t+1}(s^t, a) = V^{t+1}(s^t, a^t)$.*

Thus, the action executed last in a state has a largest value of all actions leaving the state (after the action execution).

Proof Sketch: Consider an arbitrary time t and an arbitrary action $a \in A(s^t)$. We distinguish two cases: First, $a \neq a^t$. Thus, $V^{t+1}(s^t, a) = V^t(s^t, a) \overset{\text{Lemma 1}}{\leq} \min_{a' \in A(s^t)} V^t(s^t, a') + 1 \overset{\text{action selection}}{=} V^t(s^t, a^t) + 1 = V^{t+1}(s^t, a^t)$. Second, $a = a^t$. Then, $V^{t+1}(s^t, a) = V^{t+1}(s^t, a^t)$ (trivially). Put together, $V^{t+1}(s^t, a) \leq V^{t+1}(s^t, a^t)$ for all $a \in A(s^t)$, but equality holds for at least one action (namely a^t), and the theorem follows. ∎

In the following, we use sets that can contain duplicate elements (bags). To distinguish operators on bags from operators on sets, we use an additional dot. We use the following operators: construction (description) of bags $\dot{\{\}}$, membership of an element in a bag $\dot{\in}$ (or $\dot{\ni}$), equality of bags $\doteq$, non-strict inclusion of bags $\dot{\subseteq}$ (or $\dot{\supseteq}$), union of bags $\dot{\cup}$, intersection of bags $\dot{\cap}$, and difference of bags $\dot{\setminus}$. The operators $\dot{\cup}$, $\dot{\cap}$, and $\dot{\setminus}$ have the same precedence and are left-associative. Furthermore, we always denote a bag with one element by the element itself.

We define the bags $IN1^t(s)$, $OUT1^t(s)$, $IN2^t(s)$, and $OUT2^t(s)$ inductively as follows for all times t and $s \in S$:

$$
\begin{aligned}
IN1^0(s) &:= \dot{\{}\, V^0(s', a') : s' \in S, a' \in A(s'), succ(s', a') = s \,\dot{\}} \\
OUT1^0(s) &:= \dot{\{}\, V^0(s, a) : a \in A(s) \,\dot{\}} \\
IN2^t(s) &:= IN1^t(s) \\
OUT2^t(s) &:= \begin{cases} OUT1^t(s) \mathbin{\dot{\setminus}} V^t(s^t, a^t) \mathbin{\dot{\cup}} V^{t+1}(s^t, a^t) & \text{for } s = s^t \\ OUT1^t(s) & \text{otherwise} \end{cases} \\
IN1^{t+1}(s) &:= \begin{cases} IN2^t(s) \mathbin{\dot{\setminus}} V^t(s^t, a^t) \mathbin{\dot{\cup}} V^{t+1}(s^t, a^t) & \text{for } s = s^{t+1} \\ IN2^t(s) & \text{otherwise} \end{cases} \\
OUT1^{t+1}(s) &:= OUT2^t(s)
\end{aligned}
$$

Lemma 3 *For all times t and $s \in S$,*

$$
\begin{aligned}
IN1^t(s) &\doteq \dot{\{}\ V^t(s',a') : s' \in S, a' \in A(s'), succ(s',a') = s\ \dot{\}} \\
OUT1^t(s) &\doteq \dot{\{}\ V^t(s,a) : a \in A(s)\ \dot{\}}
\end{aligned}
$$

Thus, $IN1^t(s)$ is the bag of values of all incoming actions into state s at time t, and $OUT1^t(s)$ is the bag of values of all outgoing actions from state s at time t. Similarly, $IN2^t(s) \doteq IN1^t(s)$ and $OUT2^t(s) \doteq OUT1^{t+1}(s)$ for all times t and $s \in S$.

Proof Sketch by induction on t: The lemma holds at time $t = 0$ (by definition). Assume that it holds at an arbitrary time t. Then, for all $s \in S$,

$$
\begin{aligned}
IN1^{t+1}(s) &\doteq \begin{cases} IN2^t(s) \mathbin{\dot{\setminus}} V^t(s^t,a^t) \mathbin{\dot{\cup}} V^{t+1}(s^t,a^t) & \text{for } s = s^{t+1} \\ IN2^t(s) & \text{otherwise} \end{cases} \\
&\doteq \begin{cases} IN1^t(s) \mathbin{\dot{\setminus}} V^t(s^t,a^t) \mathbin{\dot{\cup}} V^{t+1}(s^t,a^t) & \text{for } s = s^{t+1} \\ IN1^t(s) & \text{otherwise} \end{cases} \\
&\doteq \begin{cases} \dot{\{}\ V^t(s',a') : s' \in S, a' \in A(s'), succ(s',a') = s\ \dot{\}} \ldots \\ \dot{\{}\ V^t(s',a') : s' \in S, a' \in A(s'), succ(s',a') = s\ \dot{\}} \ldots \end{cases} \\
&\qquad\qquad \begin{matrix} \ldots \mathbin{\dot{\setminus}} V^t(s^t,a^t) \mathbin{\dot{\cup}} V^{t+1}(s^t,a^t) & \text{for } s = s^{t+1} \\ \ldots & \text{otherwise} \end{matrix} \\
&\doteq \dot{\{}\ V^{t+1}(s',a') : s' \in S, a' \in A(s'), succ(s',a') = s\ \dot{\}} \\
OUT1^{t+1}(s) &\doteq OUT2^t(s) \\
&\doteq \begin{cases} OUT1^t(s) \mathbin{\dot{\setminus}} V^t(s^t,a^t) \mathbin{\dot{\cup}} V^{t+1}(s^t,a^t) & \text{for } s = s^t \\ OUT1^t(s) & \text{otherwise} \end{cases} \\
&\doteq \begin{cases} \dot{\{}\ V^t(s,a) : a \in A(s)\ \dot{\}} \mathbin{\dot{\setminus}} V^t(s^t,a^t) \mathbin{\dot{\cup}} V^{t+1}(s^t,a^t) & \text{for } s = s^t \\ \dot{\{}\ V^t(s,a) : a \in A(s)\ \dot{\}} & \text{otherwise} \end{cases} \\
&\doteq \dot{\{}\ V^{t+1}(s,a) : a \in A(s)\ \dot{\}}
\end{aligned}
$$

Thus, the theorem holds at time $t+1$ as well. ■

To be able to keep our notation concise, we also define the bags $IND1^t(s)$, $OUTD1^t(s)$, $IND2^t(s)$, and $OUTD2^t(s)$ for all times t and $s \in S$:

$$
\begin{aligned}
IND1^t(s) &:= IN1^t(s) \mathbin{\dot{\setminus}} OUT1^t(s) \\
OUTD1^t(s) &:= OUT1^t(s) \mathbin{\dot{\setminus}} IN1^t(s) \\
IND2^t(s) &:= IN2^t(s) \mathbin{\dot{\setminus}} OUT2^t(s) \\
OUTD2^t(s) &:= OUT2^t(s) \mathbin{\dot{\setminus}} IN2^t(s)
\end{aligned}
$$

Note that $IND1^t(s) \doteq OUTD1^t(s) \doteq \emptyset$ iff $IN1^t(s) \doteq OUT1^t(s)$, and $IND2^t(s) \doteq OUTD2^t(s) \doteq \emptyset$ iff $IN2^t(s) \doteq OUT2^t(s)$. We call a state $s \in S$ balanced at time t if $IND1^t(s) \doteq OUTD1^t(s) \doteq \emptyset$. This means that, for every state, the number of incoming actions with value n equals the number of outgoing actions with value n for all values n.

Lemma 4 *For all times t, (A) and (B) hold, where*

(A) either

(a) $s^t = s_{start}$, and $IND1^t(s) \doteq OUTD1^t(s) \doteq \emptyset$ for all $s \in S$ (i.e. all states are balanced)

or

(b) there exist states $s_1, s_2, \ldots, s_k \in S$ and an integer y with $y \geq k-1 \geq 1$, $s_1 = s_{start}$, $s_k = s^t$, $IND1^t(s_1) \doteq y-1$ and $OUTD1^t(s_1) \doteq y$, $IND1^t(s_i) \doteq \dot{\{}\ y-i, y-i+2\ \dot{\}}$ and $OUTD1^t(s_i) \doteq \dot{\{}\ y-i+1, y-i+1\ \dot{\}}$ for $i = 2, 3, \ldots, k-1$, $IND1^t(s_k) \doteq y-k+2$ and $OUTD1^t(s_k) \doteq y-k+1$, and finally $IND1^t(s) \doteq OUTD1^t(s) \doteq \emptyset$ for $s \notin \{s_1, s_2, \ldots, s_k\}$. (Note that case (b) implies that $k \geq 2$ and $s_1, s_2, \ldots, s_k$ are pairwise different, e.g. $s^t = s_k \neq s_{start}$.)

(B) there exist states $\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_{\bar{k}} \in S$ and an integer $\bar{y}$ with $\bar{y} \geq \bar{k} \geq 1$, $V^t(s^t, a^t) = \bar{y} - \bar{k}$, $\bar{s}_1 = s_{start}$, $IND2^t(\bar{s}_1) \doteq \bar{y} - 1$ and $OUTD2^t(\bar{s}_1) \doteq \bar{y}$, $IND2^t(\bar{s}_i) \doteq \dot{\{}\ \bar{y}-i, \bar{y}-i+2\ \dot{\}}$ and $OUTD2^t(\bar{s}_i) \doteq \dot{\{}\ \bar{y}-i+1, \bar{y}-i+1\ \dot{\}}$ for $i = 2, 3, \ldots, \bar{k}$, and finally $IND2^t(s) \doteq OUTD2^t(s) \doteq \emptyset$ for $s \notin \{\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_{\bar{k}}\}$. (Note that case (B) implies that $\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_k$ are pairwise different.)

Proof Sketch by induction on t:

- (A) holds at time $t = 0$: $s^0 = s_{start}$, and $V^0(s, a) = 0$ for all $s \in S$ and $a \in A(s)$. Since the graph is Eulerian, (a) holds.
- Assume that (A) holds at an arbitrary time t. We show that (B) holds at time t as well. Note that, if $s \neq s^t$, then $IN2^t(s) \doteq IN1^t(s)$ and $OUT2^t(s) \doteq OUT1^t(s)$ and therefore $IND2^t(s) \doteq IN2^t(s) \dot{\setminus} OUT2^t(s) \doteq IN1^t(s) \dot{\setminus} OUT1^t(s) \doteq IND1^t(s)$ and $OUTD2^t(s) \doteq OUT2^t(s) \dot{\setminus} IN2^t(s) \doteq OUT1^t(s) \dot{\setminus} IN1^t(s) \doteq OUTD1^t(s)$. We distinguish two cases to determine $IND2^t(s)$ and $OUTD2^t(s)$ for $s = s^t$:
  - First, (a) holds at time t. We show that (B) holds at time t with $\bar{k} = 1$, $\bar{y} = V^t(s^t, a^t) + 1$, and $\bar{s}_1 = s_{start} = s^t$. Obviously, $\bar{y} \geq \bar{k} \geq 1$ and $V^t(s^t, a^t) = \bar{y} - \bar{k}$. Define $X := IN1^t(s^t) \dot{\cap} OUT1^t(s^t)$. Then, $IN2^t(s^t) \doteq IN1^t(s^t) \doteq X \dot{\cup} IND1^t(s^t) \doteq X \dot{\cup} \emptyset \doteq X$ and $OUT2^t(s^t) \doteq OUT1^t(s^t) \dot{\setminus} V^t(s^t, a^t) \dot{\cup} V^{t+1}(s^t, a^t) \doteq X \dot{\cup} OUTD1^t(s^t) \dot{\setminus} V^t(s^t, a^t) \dot{\cup} V^{t+1}(s^t, a^t) \doteq X \dot{\cup} \emptyset \dot{\setminus} V^t(s^t, a^t) \dot{\cup}$

$V^{t+1}(s^t,a^t) \doteq X \mathbin{\dot\setminus} V^t(s^t,a^t) \dot\cup V^{t+1}(s^t,a^t)$. Since $V^t(s^t,a^t) \dot\in OUT1^t(s^t) \doteq X \dot\cup OUTD1^t(s^t) \doteq X \dot\cup \emptyset \doteq X$ and $V^{t+1}(s^t,a^t) \neq V^t(s^t,a^t)$, it follows that $IND2^t(\bar{s}_1) \doteq IND2^t(s^t) \doteq IN2^t(s^t) \mathbin{\dot\setminus} OUT2^t(s^t) \doteq X \mathbin{\dot\setminus} (X \mathbin{\dot\setminus} V^t(s^t,a^t) \dot\cup V^{t+1}(s^t,a^t)) \doteq V^t(s^t,a^t) \doteq \bar{y}-\bar{k} \doteq \bar{y}-1$. Also, $OUTD2^t(\bar{s}_1) \doteq OUTD2^t(s^t) \doteq OUT2^t(s^t) \mathbin{\dot\setminus} IN2^t(s^t) \doteq (X \mathbin{\dot\setminus} V^t(s^t,a^t) \dot\cup V^{t+1}(s^t,a^t)) \mathbin{\dot\setminus} X \doteq V^{t+1}(s^t,a^t) \doteq V^t(s^t,a^t)+1 \doteq \bar{y}-\bar{k}+1 \doteq \bar{y}$. For $s \neq \bar{s}_1$ (i.e. $s \neq s^t$), $IND2^t(s) \doteq IND1^t(s) \doteq \emptyset$ and $OUTD2^t(s) \doteq OUTD1^t(s) \doteq \emptyset$.

– Second, (b) holds at time t. Then, there exists an action $a \in A(s^t)$ with $V^t(s^t,a) = y-k+1$, because $OUT1^t(s^t) \doteq OUT1^t(s_k) \dot\supseteq OUTD1^t(s_k) \doteq y-k+1$. Then, $y-k = V^t(s^t,a) - 1 \overset{\text{Lemma 1}}{\leq} \min_{a' \in A(s^t)} V^t(s^t,a') \overset{\text{action selection}}{=} V^t(s^t,a^t) \overset{\text{action selection}}{=} \min_{a' \in A(s^t)} V^t(s^t,a') \leq V^t(s^t,a) = y-k+1$. Thus, either $V^t(s^t,a^t) = y-k$ or $V^t(s^t,a^t) = y-k+1$. Consequently, we distinguish two subcases:

  * First, $V^t(s^t,a^t) = y-k$. We show that (B) holds at time t with $\bar{k} = k$, $\bar{y} = y$, and $\bar{s}_i = s_i$ for $1 \leq i \leq \bar{k} = k$. Obviously, $0 \leq V^t(s^t,a^t) \overset{\text{assumption}}{=} y-k = \bar{y}-\bar{k}$ and thus $\bar{y} \geq \bar{k} = k \geq 2 \geq 1$, which implies $\bar{y} \geq \bar{k} \geq 1$. Furthermore, $\bar{s}_1 = s_1 = s_{start}$. Define $X := IN1^t(s^t) \dot\cap OUT1^t(s^t)$. Then, $IN2^t(s^t) \doteq IN1^t(s^t) \doteq X \dot\cup IND1^t(s^t) \doteq X \dot\cup IND1^t(s_k) \doteq X \dot\cup y-k+2$ and $OUT2^t(s^t) \doteq OUT1^t(s^t) \mathbin{\dot\setminus} V^t(s^t,a^t) \dot\cup V^{t+1}(s^t,a^t) \doteq X \dot\cup OUTD1^t(s^t) \mathbin{\dot\setminus} y-k \dot\cup y-k+1 \doteq X \dot\cup OUTD1^t(s_k) \mathbin{\dot\setminus} y-k \dot\cup y-k+1 \doteq X \dot\cup y-k+1 \mathbin{\dot\setminus} y-k \dot\cup y-k+1 \doteq X \mathbin{\dot\setminus} y-k \dot\cup \dot\{\ y-k+1, y-k+1\ \dot\}$. Note that $y-k \dot\in X$, because $V^t(s^t,a^t) \overset{\text{assumption}}{=} y-k$ and thus $y-k \dot\in OUT1^t(s^t) \doteq X \dot\cup OUTD1^t(s^t) \doteq X \dot\cup OUTD1^t(s_k) \doteq X \dot\cup y-k+1$. Consequently, $IND2^t(s^t) \doteq IN2^t(s^t) \mathbin{\dot\setminus} OUT2^t(s^t) \doteq \dot\{\ y-k, y-k+2\ \dot\} \doteq \dot\{\ \bar{y}-\bar{k}, \bar{y}-\bar{k}+2\ \dot\}$ and $OUTD2^t(s^t) \doteq OUT2^t(s^t) \mathbin{\dot\setminus} IN2^t(s^t) \doteq \dot\{\ y-k+1, y-k+1\ \dot\} \doteq \dot\{\ \bar{y}-\bar{k}+1, \bar{y}-\bar{k}+1\ \dot\}$ for $s^t = s_k = \bar{s}_{\bar{k}}$. It is easy to show that the theorem also holds for $s \neq s^t$, using $IND2^t(s) \doteq IND1^t(s)$ and $OUTD2^t(s) \doteq OUTD1^t(s)$ together with $\bar{y} = y$.
  * Second, $V^t(s^t,a^t) = y-k+1$. We show that (B) holds at time t with $\bar{k} = k-1$, $\bar{y} = y$, and $\bar{s}_i = s_i$ for $1 \leq i \leq \bar{k} = k-1$. Obviously, $\bar{y} \geq \bar{k} \geq 1$ (since $y \geq k-1 \geq 1$), $V^t(s^t,a^t) \overset{\text{assumption}}{=} y-k+1 = \bar{y}-\bar{k}$, and $\bar{s}_1 = s_1 = s_{start}$. Define $X := IN1^t(s^t) \dot\cap OUT1^t(s^t)$. Then, $IN2^t(s^t) \doteq IN1^t(s^t) \doteq X \dot\cup IND1^t(s^t) \doteq X \dot\cup IND1^t(s_k) \doteq X \dot\cup y-k+2$ and $OUT2^t(s^t) \doteq OUT1^t(s^t) \mathbin{\dot\setminus} V^t(s^t,a^t) \dot\cup V^{t+1}(s^t,a^t) \doteq OUT1^t(s_k) \mathbin{\dot\setminus} y-k+1 \dot\cup y-k+2 \doteq X \dot\cup OUTD1^t(s_k) \mathbin{\dot\setminus}$

$y-k+1 \mathbin{\dot\cup} y-k+2 \doteq X \mathbin{\dot\cup} y-k+1 \mathbin{\dot\setminus} y-k+1 \mathbin{\dot\cup} y-k+2 \doteq X \mathbin{\dot\cup} y-k+2$. Consequently, $IND2^t(s^t) \doteq IN2^t(s^t) \mathbin{\dot\setminus} OUT2^t(s^t) \doteq \emptyset$ and $OUTD2^t(s^t) \doteq OUT2^t(s^t) \mathbin{\dot\setminus} IN2^t(s^t) \doteq \emptyset$ for $s^t = s_k \notin \{s_1, \ldots, s_{k-1}\} = \{\bar{s}_1, \ldots, \bar{s}_{\bar{k}}\}$. It is easy to show that the theorem also holds for $s \neq s^t$, using $IND2^t(s) \doteq IND1^t(s)$ and $OUTD2^t(s) \doteq OUTD1^t(s)$ together with $\bar{y} = y$.

- Assume that (B) holds at an arbitrary time t. We show that (A) holds at time $t+1$. Note that, if $s \neq s^{t+1}$, then $IN1^{t+1}(s) \doteq IN2^t(s)$ and $OUT1^{t+1}(s) \doteq OUT2^t(s)$ and therefore $IND1^{t+1}(s) \doteq IN1^{t+1}(s) \mathbin{\dot\setminus} OUT1^{t+1}(s) \doteq IN2^t(s) \mathbin{\dot\setminus} OUT2^t(s) \doteq IND2^t(s)$ and $OUTD1^{t+1}(s) \doteq OUT1^{t+1}(s) \mathbin{\dot\setminus} IN1^{t+1}(s) \doteq OUT2^t(s) \mathbin{\dot\setminus} IN2^t(s) \doteq OUTD2^t(s)$. We distinguish two cases to determine $IND1^{t+1}(s)$ and $OUTD1^{t+1}(s)$ for $s = s^{t+1}$:
  - First, $s^{t+1} \notin \{\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_{\bar{k}}\}$. We show that (b) holds at time $t+1$ with $k = \bar{k}+1$, $y = \bar{y}$, $s_i = \bar{s}_i$ for $1 \leq i \leq k-1 = \bar{k}$, and $s_k = s^{t+1}$. Obviously, $y \geq k-1 \geq 1$ (since $y = \bar{y} \geq \bar{k} = k-1 = \bar{k} \geq 1$) and $s_1 = \bar{s}_1 = s_{start}$. Define $X := IN2^t(s^{t+1}) \mathbin{\dot\cap} OUT2^t(s^{t+1})$. Then, $IN1^{t+1}(s^{t+1}) \doteq IN2^t(s^{t+1}) \mathbin{\dot\setminus} V^t(s^t, a^t) \mathbin{\dot\cup} V^{t+1}(s^t, a^t) \doteq IN2^t(s^{t+1}) \mathbin{\dot\setminus} \bar{y}-\bar{k} \mathbin{\dot\cup} \bar{y}-\bar{k}+1 \doteq X \mathbin{\dot\cup} IND2^t(s^{t+1}) \mathbin{\dot\setminus} \bar{y}-\bar{k} \mathbin{\dot\cup} \bar{y}-\bar{k}+1 \doteq X \mathbin{\dot\cup} \emptyset \mathbin{\dot\setminus} \bar{y}-\bar{k} \mathbin{\dot\cup} \bar{y}-\bar{k}+1 \doteq X \mathbin{\dot\setminus} \bar{y}-\bar{k} \mathbin{\dot\cup} \bar{y}-\bar{k}+1$ and $OUT1^{t+1}(s^{t+1}) \doteq OUT2^t(s^{t+1}) \doteq X \mathbin{\dot\cup} OUTD2^t(s^{t+1}) \doteq X \mathbin{\dot\cup} \emptyset \doteq X$. Note that $\bar{y}-\bar{k} \mathrel{\dot\in} X$, because $V^t(s^t, a^t) = \bar{y}-\bar{k}$ and thus $\bar{y}-\bar{k} \mathrel{\dot\in} IN1^t(s^{t+1}) \doteq IN2^t(s^{t+1}) \doteq X \mathbin{\dot\cup} IND2^t(s^{t+1}) \doteq X \mathbin{\dot\cup} \emptyset \doteq X$. Consequently, $IND1^{t+1}(s^{t+1}) \doteq IN1^{t+1}(s^{t+1}) \mathbin{\dot\setminus} OUT1^{t+1}(s^{t+1}) \doteq \bar{y}-\bar{k}+1 \doteq y-k+2$ and $OUTD1^{t+1}(s^{t+1}) \doteq OUT1^{t+1}(s^{t+1}) \mathbin{\dot\setminus} IN1^{t+1}(s^{t+1}) \doteq \bar{y}-\bar{k} \doteq y-k+1$ for $s^{t+1} = s_k$. It is easy to show that the theorem also holds for $s \neq s^{t+1}$, using $IND1^{t+1}(s) \doteq IND2^t(s)$ and $OUTD1^{t+1}(s) \doteq OUTD2^t(s)$ together with $y = \bar{y}$.
  - Second, $s^{t+1} \in \{\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_{\bar{k}}\}$. We show that $s^{t+1} = \bar{s}_{\bar{k}}$ by contradiction. Assume that $s^{t+1} = \bar{s}_i$ for some $i < \bar{k}$ and define $X := IN2^t(s^{t+1}) \mathbin{\dot\cap} OUT2^t(s^{t+1})$. Then, there exists an action $a \in A(s^{t+1})$ with $V^{t+1}(s^{t+1}, a) = \bar{y}-\bar{k}$, because $V^t(s^t, a^t) = \bar{y}-\bar{k}$ and thus $\bar{y}-\bar{k} \mathrel{\dot\in} IN1^t(s^{t+1}) \doteq IN2^t(s^{t+1}) \doteq X \mathbin{\dot\cup} IND2^t(s^{t+1})$; it must be that $\bar{y}-\bar{k} \mathrel{\dot\in} X \mathrel{\dot\subseteq} OUT2^t(s^{t+1}) \doteq OUT1^{t+1}(s^{t+1})$, since $\bar{y}-\bar{k} \mathrel{\dot\notin} IND2^t(\bar{s}_i) = IND2^t(s^{t+1})$. There also exists an action $a' \in A(s^{t+1})$ with $V^{t+1}(s^{t+1}, a') = \bar{y}-i+1$, because $OUT1^{t+1}(s^{t+1}) \doteq OUT2^t(s^{t+1}) \mathrel{\dot\supseteq} OUTD2^t(s^{t+1}) \doteq OUTD2^t(\bar{s}_i) \mathrel{\dot\ni} \bar{y}-i+1$. Put together, $V^{t+1}(s^{t+1}, a') = \bar{y}-i+1 > \bar{y}-\bar{k}+1 = V^{t+1}(s^{t+1}, a)+1$ (since $i < \bar{k}$). This, however, is a contradiction to Lemma 1, which asserts that $V^{t+1}(s^{t+1}, a') \leq \min_{a'' \in A(s^{t+1})} V^{t+1}(s^{t+1}, a'')+1 \leq V^{t+1}(s^{t+1}, a)+1$. It follows that

$s^{t+1} = \bar{s}_{\bar{k}}$. We distinguish two subcases:

* First, $\bar{k} = 1$. We show that (a) holds at time $t+1$. Obviously, $s^{t+1} = \bar{s}_{\bar{k}} = \bar{s}_1 = s_{start}$. Define $X := IN2^t(s^{t+1}) \mathbin{\dot\cap} OUT2^t(s^{t+1})$. Then, $IN1^{t+1}(s^{t+1}) \doteq IN2^t(s^{t+1}) \mathbin{\dot\setminus} V^t(s^t, a^t) \mathbin{\dot\cup} V^{t+1}(s^t, a^t) \doteq IN2^t(s^{t+1}) \mathbin{\dot\setminus} \bar{y} - \bar{k} \mathbin{\dot\cup} \bar{y} - \bar{k} + 1 \doteq X \cup IND2^t(s^{t+1}) \mathbin{\dot\setminus} \bar{y} - 1 \mathbin{\dot\cup} \bar{y} \doteq X \cup IND2^t(\bar{s}_1) \mathbin{\dot\setminus} \bar{y} - 1 \mathbin{\dot\cup} \bar{y} \doteq X \mathbin{\dot\cup} \bar{y} - 1 \mathbin{\dot\setminus} \bar{y} - 1 \mathbin{\dot\cup} \bar{y} \doteq X \mathbin{\dot\cup} \bar{y}$ and $OUT1^{t+1}(s^{t+1}) \doteq OUT2^t(s^{t+1}) \doteq X \mathbin{\dot\cup} OUTD2^t(s^{t+1}) \doteq X \mathbin{\dot\cup} OUTD2^t(\bar{s}_1) \doteq X \mathbin{\dot\cup} \bar{y}$. Consequently, $IND1^{t+1}(s^{t+1}) \doteq IN1^{t+1}(s^{t+1}) \mathbin{\dot\setminus} OUT1^{t+1}(s^{t+1}) \doteq \emptyset$ and $OUTD1^{t+1}(s^{t+1}) \doteq OUT1^{t+1}(s^{t+1}) \mathbin{\dot\setminus} IN1^{t+1}(s^{t+1}) \doteq \emptyset$ for $s^{t+1} = \bar{s}_1$. For $s \neq s^{t+1}$, $IND1^{t+1}(s) \doteq IND2^t(s) \doteq \emptyset$ and $OUTD1^{t+1}(s) \doteq OUTD2^t(s) \doteq \emptyset$.
* Second, $\bar{k} > 1$. We show that (b) holds at time $t+1$ with $k = \bar{k}$, $y = \bar{y}$, and $s_i = \bar{s}_i$ for $1 \leq i \leq k = \bar{k}$. Obviously, $y \geq k - 1 \geq 1$ (since $y = \bar{y} \geq \bar{k} = k = \bar{k} > 1$), $s_1 = \bar{s}_1 = s_{start}$, and $s_k = \bar{s}_{\bar{k}} = s^{t+1}$. Define $X := IN2^t(s^{t+1}) \mathbin{\dot\cap} OUT2^t(s^{t+1})$. Then, $IN1^{t+1}(s^{t+1}) \doteq IN2^t(s^{t+1}) \mathbin{\dot\setminus} V^t(s^t, a^t) \cup V^{t+1}(s^t, a^t) \doteq IN2^t(s^{t+1}) \mathbin{\dot\setminus} \bar{y} - \bar{k} \cup \bar{y} - \bar{k} + 1 \doteq X \mathbin{\dot\cup} IND2^t(s^{t+1}) \mathbin{\dot\setminus} \bar{y} - \bar{k} \cup \bar{y} - \bar{k} + 1 \doteq X \mathbin{\dot\cup} IND2^t(\bar{s}_{\bar{k}}) \mathbin{\dot\setminus} \bar{y} - \bar{k} \cup \bar{y} - \bar{k} + 1 \doteq X \mathbin{\dot\cup} \dot{\{}\, \bar{y} - \bar{k}, \bar{y} - \bar{k} + 2 \,\dot{\}} \mathbin{\dot\setminus} \bar{y} - \bar{k} \cup \bar{y} - \bar{k} + 1 \doteq X \mathbin{\dot\cup} \dot{\{}\, \bar{y} - \bar{k} + 1, \bar{y} - \bar{k} + 2 \,\dot{\}}$ and $OUT1^{t+1}(s^{t+1}) \doteq OUT2^t(s^{t+1}) \doteq X \mathbin{\dot\cup} OUTD2^t(s^{t+1}) \doteq X \mathbin{\dot\cup} OUTD2^t(\bar{s}_{\bar{k}}) \doteq X \mathbin{\dot\cup} \dot{\{}\, \bar{y} - \bar{k} + 1, \bar{y} - \bar{k} + 1 \,\dot{\}}$. Consequently, $IND1^{t+1}(s^{t+1}) \doteq IN1^{t+1}(s^{t+1}) \mathbin{\dot\setminus} OUT1^{t+1}(s^{t+1}) \doteq \bar{y} - \bar{k} - 2 \doteq y - k + 2$ and $OUTD1^{t+1}(s^{t+1}) \doteq OUT1^{t+1}(s^{t+1}) \mathbin{\dot\setminus} IN1^{t+1}(s^{t+1}) \doteq \bar{y} - \bar{k} + 1 \doteq y - k + 1$ for $s^{t+1} = \bar{s}_{\bar{k}}$ (note that $s^{t+1} \neq \bar{s}_1$). It is easy to show that the theorem also holds for $s \neq s^{t+1}$, using $IND1^{t+1}(s) \doteq IND2^t(s)$ and $OUTD1^{t+1}(s) \doteq OUTD2^t(s)$ together with $y = \bar{y}$. ■

Lemma 5 *For all times t, $s \in S$, and $a \in A(s)$, $V^t(s, a) \leq \max_{a' \in A(s_{start})} V^t(s_{start}, a')$.*

Thus, there is always an action that leaves the start state and has a largest value of all actions.

Proof Sketch by induction on t: The lemma holds at time $t = 0$, since $V^0(s, a) = 0$ for all $s \in S$ and $a \in A(s)$. Assume that the lemma holds at an arbitrary time t, and consider arbitrary $s \in S$ and $a \in A(s)$. We distinguish two cases: First, $s \neq s^t$ or $a \neq a^t$. Then, $V^{t+1}(s, a) = V^t(s, a) \overset{\text{assumption}}{\leq} \max_{a' \in A(s_{start})} V^t(s_{start}, a') \overset{\text{monotonicity}}{\leq} \max_{a' \in A(s_{start})} V^{t+1}(s_{start}, a')$. Second, $s = s^t$ and $a = a^t$. We distinguish two subcases: First, $s = s_{start}$. Then,

$V^{t+1}(s,a) = V^{t+1}(s_{start},a) \leq \max_{a' \in A(s_{start})} V^{t+1}(s_{start},a')$ (trivially). Second, $s \neq s_{start}$. Then, consider the variables from Lemma 4(B) at time t. There exists an action $a' \in A(s_{start})$ with $V^{t+1}(s_{start},a') = \bar{y}$, because $OUT1^{t+1}(s_{start}) \doteq OUT2^t(s_{start}) \doteq OUT2^t(\bar{s}_1) \dot{\supseteq} OUTD2^t(\bar{s}_1) \doteq \bar{y}$. Then, $V^{t+1}(s,a) = V^{t+1}(s^t,a^t) = V^t(s^t,a^t)+1 = (\bar{y}-\bar{k})+1 \overset{\bar{k}\geq 1}{\leq} \bar{y} = V^{t+1}(s_{start},a') \leq \max_{a'' \in A(s_{start})} V^{t+1}(s_{start},a'')$, and the lemma holds at time $t+1$ as well. ∎

Theorem 9 *The complexity of edge counting is at most $e \times gd(s_{start}) - gd(s_{start})^2$ action executions in undirected or Eulerian state spaces.*

Comment: The proof of Theorem 8 shows that this bound is tight.

Proof Sketch: If $s_{start} \in G$, then $gd(s_{start}) = 0$ and the theorem holds, since the goal state is reached without any action executions. Assume that $s_{start} \notin G$. According to Theorem 4, edge counting does reach a goal state eventually. In the following, we analyze how many action executions it requires at most.

Consider the latest time t with $s^t = s_{start}$. According to Lemma 4(a), all states (including s_{start}) are balanced at time t. Now consider a shortest path from s_{start} to a closest goal state. The last action on that path has never been executed and thus has value 0 at time t. Consequently, the largest value of any action that leaves the last non-goal state s on the path is 0 if no other action leaves s, otherwise – according to Lemma 1 – the largest value of any action leaving s is at most 1. Since s is balanced at time t, the largest value of any action entering s is 0 in the former case and at most 1 in the latter case. Thus, the value of the action that precedes the last action on the path is 0 or at most 1, respectively, etc. Finally, the value $V^t(s_{start},a)$ of the first action a on the path is at most x where x is the number of intermediate states on the path (i.e. not including the start and goal state) that have two or more outgoing actions. $V^t(s_{start},a) \leq x \leq gd(s_{start})-1$, because the path has length $gd(s_{start})$ and therefore $gd(s_{start})-1$ intermediate states. $V^t(s_{start},a) \leq x \leq e - gd(s_{start}) - 1$, because each of the $gd(s_{start})+1$ states on the path (including the start and goal state) has at least one outgoing action. Each state with two or more outgoing actions needs at least one more action of the remaining $e - gd(s_{start}) - 1$ actions. Put together, $\min_{a' \in A(s_{start})} V^t(s_{start},a') \leq V^t(s_{start},a) \leq x \leq \min(gd(s_{start}) - 1, e - gd(s_{start}) - 1)$.

Now consider the time $t' > t$ when edge counting reaches a goal state for the first time. According to Lemma 4(b), there exist pairwise different states $s_1, s_2, \ldots, s_k \in S$ with $s_1 = s_{start}$, $s_k = s^{t'}$, $OUTD1^{t'}(s_1) \doteq y$, $OUTD1^{t'}(s_i) \doteq \{ y-i+1, y-i+1 \dot{\}}$ for $i = 2,3,\ldots,k-1$, and $OUTD1^{t'}(s_k) \doteq y-k+1$. The values k (with $k > 1$) and y are calculated as follows: At time t', all actions entering and leaving the reached goal state $s^{t'}$ have value 0, with the exception of the action with which it was entered, which has value 1. Thus, $0 \doteq OUTD1^{t'}(s^{t'}) \doteq OUTD1^{t'}(s_k) \doteq y-k+1$ and consequently $k = y+1$. Similarly, $y \doteq OUTD1^{t'}(s_1) \doteq OUTD1^{t'}(s_{start}) \doteq OUTD1^{t+1}(s_{start}) \doteq V^{t+1}(s_{start}, a^t)$ and consequently $y = V^{t+1}(s_{start}, a^t)$, since s_{start} was balanced at time t and edge counting never entered s_{start} again after time t. $y = V^{t+1}(s_{start}, a^t) = V^{t+1}(s^t, a^t) = V^t(s^t, a^t){+}1 = V^t(s_{start}, a^t){+}1 \leq \min_{a' \in A(s_{start})} V^t(s_{start}, a'){+}1 \overset{\text{previous paragraph}}{\leq} \min(gd(s_{start})-1, e-gd(s_{start})-1)+1 = \min(gd(s_{start}), e-gd(s_{start}))$.

To summarize, at time t', there is one action (in state s_1) with value y, there are two actions (in state s_i) with value $y-i+1$ for $i = 2, 3, \ldots, k-1 = y$, and there is one action (in state $s_k = s_{y+1}$) with value y-k+1=0. Since the states $s_1, s_2, \ldots, s_k$ are pairwise different, this accounts for $2(y-1)+2 = 2y$ actions. The values of the remaining $e-2y$ actions can be at most (see Lemma 5) $\max_{a' \in A(s_{start})} V^{t'}(s_{start}, a') = \max_{a' \in A(s_{start})} V^{t+1}(s_{start}, a') = \max_{a' \in A(s^t)} V^{t+1}(s^t, a') \overset{\text{Lemma 2}}{=} V^{t+1}(s^t, a^t) = V^{t+1}(s_{start}, a^t) = y$ at time t', since edge counting never entered s_{start} again after time t. It follows that the sum of the values of all actions at time t' is at most

$$y + 2\sum_{i=2}^{y}(y-i+1) + 0 + (e-2y)y = e \times y - y^2$$

This expression is maximized for $y = e/2$. However, there is the restriction on y that $y \leq \min(gd(s_{start}), e-gd(s_{start}))$. If $gd(s_{start}) \leq e/2$, then $gd(s_{start}) \leq e/2 \leq e - gd(s_{start})$ and $y = gd(s_{start})$ is optimal. If $gd(s_{start}) \geq e/2$, then $e - gd(s_{start}) \leq e/2 \leq gd(s_{start})$ and $y = e - gd(s_{start})$ is optimal. In both cases, the sum of the values of all actions at time t' is at most $e \times y - y^2 \leq e \times gd(s_{start}) - gd(s_{start})^2$. The theorem follows because the total number of action executions corresponds to the sum of the values of all actions at time t'. ∎