Using Swamps to Improve Optimal Pathfinding

Nir Pochter, Aviv Zohar, and Jeffrey S. Rosenschein
School of Engineering and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel
{nirp, avivz, jeff}@cs.huji.ac.il

Abstract

In a variety of domains, such as computer games and robotics, many shortest paths have to be found quickly in real time. We address the problem of quickly finding shortest paths in large known graphs. We propose a method that relies on identifying areas that tend to be searched needlessly (areas we call *swamp-regions*), and exploits this knowledge to improve search. The method requires storing only a few bits in memory for each node of the graph, and reduces search cost drastically, while still finding optimal paths. Our method is independent of the heuristics used in the search, and of the search algorithm. We present experimental results that support our claims, and provide an anytime algorithm for the pre-processing stage that identifies swamp-regions.

Introduction

Many real-time applications search for shortest paths in known graphs. Examples include strategy games where multiple units traverse a large board, as well as robotics applications where robots are required to navigate, planning their path through some environment. The frequency that the system has to search for paths can strain its resources and damage performance.

Heuristics are commonly used to improve the running time of search over graphs. While heuristic algorithms, such as A*, usually succeed in improving search cost when compared to uninformed search algorithms, there is still much room for improvement. In this paper, we introduce a method that prunes the search graph by removing areas where search is usually wasted; this pruning thus lowers the overall search cost. Our method guarantees that the paths that are found are optimal, even after the graph has been pruned.

First, let us motivate our discussion regarding "difficult search areas". Consider the map given in Figure 1; in this example, algorithms such as A* (Hart, Nilsson, & Raphael July 1968) (with a good heuristic) can search very efficiently in some areas of the map, while being very inefficient in other areas. Figure 1 shows the nodes that are expanded during a search from node S to node T, as carried out by the A* algorithm, using a Manhattan distance heuristic on a four-neighbor, two-dimensional grid. Note that while the optimal path that is eventually found is quite short, the number of expanded nodes is significantly larger. Many nodes are expanded inside the cup-shaped region, while the final path does not pass through any node in that region. In fact, any shortest path that does not start inside the cup or end in it, will never pass through any node within it.

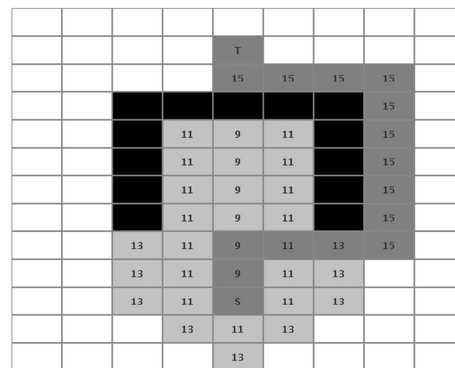


Figure 1: Nodes expanded during an A* search from node S to node T. Obstacles are marked in black, the expanded nodes are marked in gray, and the f value used during the search is noted for each one. The path that is eventually found is marked in darker gray.

Our approach will be to automatically identify areas such as the cup, which we will call *swamp-regions*, and store information about them in the graph, without wasting too much memory. Then, while searching for shortest paths between two nodes of the graph, we can block the search as it tries to unnecessarily enter those regions. We will present anytime algorithms for the pre-processing stage in which we locate swamp-regions in a grid; i.e., the algorithms give better results the longer they run. The detection process can thus be run in the background, using spare processing time to improve the results of future searches in the graph (freeing more processing time in the future, when it may be more scarce). Our algorithms are also applicable in cases where the grid changes slowly, as we are able to quickly update the swamp-regions to reflect minor changes in the environment.

We empirically evaluated our method on 2D four-neighbor grids with randomly placed obstacles, where

Copyright © 2008, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

search is performed using the A* algorithm with an admissible consistent heuristic. The results demonstrate the usefulness of our approach and provide information regarding the efficiency of our method.

The rest of the paper is organized as follows. We begin by briefly reviewing related work, and then turn to formally defining swamps on general graphs, and proving some of their properties. We then explain how to exploit information about swamp-regions during the search to obtain shortest paths while expanding fewer nodes. Next, we present an algorithm to detect swamps on a 2D four-neighbor grid, and prove its correctness. We then present experimental results that support the claim that using our algorithms on four-neighbor grids significantly reduces the search cost. We conclude by discussing future work.

Related Work

Much research has been carried out in the field of artificial intelligence to improve the speed of search operations on graphs under various circumstances, while not consuming a large amount of memory. A* (Hart, Nilsson, & Raphael July 1968) and IDA* (Korf 1985) are widely used, where A* is usually faster but can consume more memory than IDA*.

Several methods, such as (Sturtevant 2007), (Botea, Müller, & Schaeffer 2004) and (Sturtevant & Buro 2005), use graph abstractions to increase the speed of search. Those methods pre-process a grid and build an abstract representation of the search graph, sometimes at multiple levels. The search is then done in the abstract graph, which is smaller, and is refined into the original graph. These methods have been shown to work well on large graphs, though they do not guarantee shortest paths, and sometimes require a path-smoothing phase after the path refinement in order to get good results.

Another approach is to use previous searches to improve new search performance. LPRA* (Korf 1990), (Koenig 2004) and RTAA (Koenig & Likhachev 2006) search with limited look-ahead, and update the heuristic of the nodes visited. These approaches solve the first move delay problem, but pay a price since the paths they find are not guaranteed to be shortest paths, and convergence time may be long.

LPA* (Koenig, Likhachev, & Furcy 2004) and D* lite (Koenig & Likhachev 2002) reuse previous search information when the environment is dynamic; thus the path found in previous searches might no longer be passable, or might no longer be the optimal path, due to a change in the map. Those algorithms use the previous search information to recalculate the path, either from the original start point (LPA*) or from the current position of the agent (D* lite), and usually perform better than beginning a new A* search.

Exploiting swamps implies searching in a smaller set of available nodes, and can therefore be of benefit to all the algorithms mentioned above and many others; it does not compete with them. Our algorithm just adds a pre-processing stage that should be executed once per graph.

Swamps

We now define swamp regions. Intuitively, a swamp is an area in the graph such that any shortest path that passes through it either starts or ends inside that area, or has an alternative shortest path that does not pass through.[1] We define this notion more formally below.

Definition 1. *A swamp-set $\mathcal{S}$ in an undirected graph $G = (V, E)$ is a group of nodes $\mathcal{S} \subseteq V$ such that any two nodes v_1, v_2 which are not part of $\mathcal{S}$ have a shortest path that does not pass through $\mathcal{S}$:*
For each $v_1, v_2 \notin \mathcal{S}$, there exists a shortest path $P_{1,2}$ that connects v_1 and v_2 such that $P_{1,2} \cap \mathcal{S} = \emptyset$.

Note that a swamp-set is not necessarily a connected component in the graph. We shall use the term *swamp-region* to denote a connected component that is a swamp-set.

Definition 2. *A swamp-region $\mathcal{R}$ is a set of connected nodes that is a swamp-set.*

The next example illustrates the definition of a swamp.

Example 1. *Figure 2 demonstrates a swamp-region. Let $\mathcal{R} = \{s_1, s_2, s_3, s_4\}$. For any search from node $S \in V \setminus \mathcal{R}$ to a node $T \in V \setminus \mathcal{R}$ there exists a shortest path that does not pass through any of the nodes in $\{s_1, \ldots, s_4\}$.*

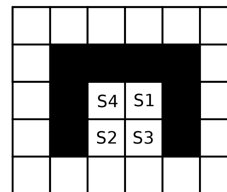


Figure 2: An example of a swamp-region. Nodes filled in black are obstacles. Nodes $\{s_1, s_2, s_3, s_4\}$ form a swamp-region.

We now define the *external boundary* of a swamp-set as follows:

Definition 3. *The external boundary of a swamp-set $\mathcal{S}$, $B(\mathcal{S})$, is the collection of nodes that are connected directly to nodes of the swamp-set but are not part of it.*

Additional Properties of Swamp-Sets

We shall now demonstrate a few properties of swamps that will later be used in our algorithm's detection and exploitation of swamps.

Our first lemma shows us that it is enough to check only paths between points on the boundary of a region in order to ensure that it is a swamp-set. This will later give us a good procedure for checking if a given set of nodes is a swamp-set, and for trimming down a region to a swamp-region.

Lemma 1. *Let $\mathcal{S}$ be a set of nodes in V. If for any two nodes on the external boundary of $\mathcal{S}$, $v_1, v_2 \in B(\mathcal{S})$, there exists a shortest path between v_1, v_2 that does not pass through $\mathcal{S}$, then $\mathcal{S}$ is a swamp-set.*

[1] A slightly more restrictive alternative is to define a swamp-set as a group of nodes that is *never* used in any shortest path. This definition has nicer properties in some sense, but yields significantly smaller swamp-sets and is thus less useful in practice.

Proof. Assume that the claim is not correct; then there exist two nodes, v_1 and v_2 that are not in $\mathcal{S}$, such that there is at least one shortest path between v_1 and v_2 that passed through $\mathcal{S}$, and no shortest path between v_1 and v_2 does not pass through $\mathcal{S}$. Since v_1 and v_2 are not in $\mathcal{S}$ any path between them that passes through $\mathcal{S}$ has to enter and leave $\mathcal{S}$. This means that it passed through at least two points in $B(\mathcal{S})$. We will mark the first such node as v_{B1} and the last as v_{B2}. According to the conditions of the lemma, there is a shortest path between v_{B1} and v_{B2} that does not pass through the group, so we can replace the part between v_{B1} and v_{B2} with this path, thus getting a shorter path between v_1 and v_2 that does not pass through $\mathcal{S}$—in contradiction to the claim. □

Our second lemma demonstrates that if a swamp-set is composed of several isolated components, then each one of them is in fact a swamp-region. We will therefore later be able to remove isolated components of a swamp without damaging the properties of the rest of the swamp-set.

Lemma 2. *Any connected component $\mathcal{R}$ that is contained in a swamp-set $\mathcal{S}$, and is isolated from the rest of the swamp (i.e., $B(\mathcal{R}) \cap \mathcal{S} = \emptyset$) is also a swamp-region.*

Proof. According to Lemma 1, it is enough to show that any two points on the boundary of $\mathcal{R}$ have a connecting shortest path that does not pass through $\mathcal{R}$. We know this is true, because by definition $B(\mathcal{R})$ consists only of obstacles or nodes that do not belong to $\mathcal{S}$. Therefore, because $\mathcal{S}$ is a swamp-set, we know that at least one shortest path between these points passes outside of $\mathcal{S}$ and therefore also outside of $\mathcal{R}$, which is a subset of the nodes of the swamp-set $\mathcal{S}$. □

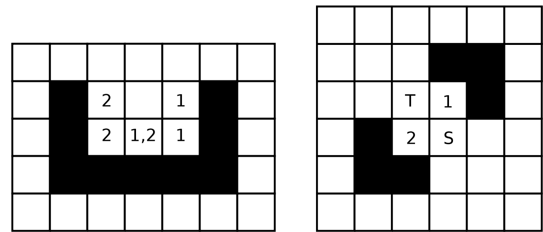
In the previous lemma we have shown that every isolated component can be broken down to swamp-regions. In fact, it is sometimes possible to further decompose each isolated component to swamp-regions. Later in the paper we will discuss the problem of detecting swamp-sets in the graph, and especially those that we can partition to many swamp-regions. Here we shall show some properties of swamps that demonstrate why this is not trivial.

Ideally it would be useful if swamps were monotonic in some way, i.e., if each subset of nodes from a swamp-set would compose a smaller swamp-set. This is not the case. In fact, even the intersection of two known swamp-sets (which is therefore contained in both) is not necessarily a swamp-set.

Lemma 3. *The intersection of two swamp-sets, $\mathcal{S}_1$ and $\mathcal{S}_2$, is not necessarily a swamp-set.*

Proof. We prove this by example. Consider the grid displayed in Figure 3(a). The group of nodes marked by 1 forms a swamp-set, and so does the group marked by 2. However, their intersection (the node that is marked $1, 2$) is not a swamp-set, as the only shortest path between the corner nodes inside the cup-shape passes through it. □

Another property that we would have found useful is to be able to unify swamp-sets, and thus locate larger ones. Even this procedure does not always succeed.



(a) An example that shows that an intersection of swamp-sets is not necessarily a swamp-set.

(b) An example that shows that the unification of swamp-sets is not necessarily a swamp-set.

Figure 3: Figures used in the proofs of Lemmas 3 and 4

Lemma 4. *The unification of two swamp-sets, $\mathcal{S}_1$ and $\mathcal{S}_2$, may not be a swamp-set.*

Proof. Again, we prove by example. Consider the grid that is displayed in Figure 3(b). The node marked by 1 forms a swamp-set if all other nodes are not swamps. The same holds for node 2. Their unification, however, is not a swamp-set, as the shortest path from S to T must pass either through node 1, or through node 2. □

Using Swamps to Decrease Search Costs

A naive approach to using a swamp-set to lower search costs is to consider them as blocked whenever a search between two nodes from outside the swamp-set is performed. The search is then performed on an effectively smaller graph, and could be expected to open fewer nodes. By the definition of swamp-sets, the path that is found is still optimal. Using this approach, more nodes are pruned from the graph when the swamp is larger. However, in these cases fewer paths will enjoy the benefits of pruning, since any arbitrary source and target nodes are less likely to be outside a large swamp-set.

We will try to increase the benefits we get from swamps by using a swamp-set that is completely partitioned into different swamp-regions. As we later show, we can construct a swamp-set such that any subset of the swamp-regions that make up this swamp-set will also constitute a swamp-set. This way, when we search between two nodes in the graph, we will consider any swamp-region that they do not belong to as blocked, and thus achieve significant savings on searches between swamp nodes as well. More formally, when searching for a path between nodes v_1 and v_2:

1. Let V be the set of vertices in the graph.
2. Let $\mathcal{S}$ be the full swamp-set that was found in the graph, and is completely partitioned into swamp-regions such that every subset of swamp-regions forms a swamp-set as well.
3. Let $\mathcal{R}_1$ be the swamp-region that v_1 belongs to, or $\emptyset$ if v_1 does not belong to any swamp-region.
4. Let $\mathcal{R}_2$ be the swamp-region that v_2 belongs to, or $\emptyset$ if v_2 does not belong to any swamp-region.
5. Search only in the nodes of $(V \setminus \mathcal{S}) \cup \mathcal{R}_1 \cup \mathcal{R}_2$.

Lemma 5. *Searching under the above conditions maintains optimality in the sense of shortest paths.*

Proof. Let us examine a search between any two arbitrary nodes, v_1 and v_2. We know from our assumption that $\mathcal{S} \setminus (\mathcal{R}_1 \cup \mathcal{R}_2)$ is a swamp-set, and therefore any search that ignores those nodes can still produce an optimal path. □

Another potential use of swamps is shown in the experimental section. We have experimentally found that when searching for paths into a swamp-region the search cost is often higher than when we search for the same path in reverse. We therefore always reverse the search when it originates outside a swamp and ends inside one.[2] We believe the explanation for this phenomenon is that swamps are usually near large obstacles in the graph. When searching from a node that is next to an obstacle to a node that is far away on the opposite side of it, the search expands fewer nodes than the reversed search.

In the next section, we will show how to find a swamp-set that can be partitioned into swamp-regions in a manner that will satisfy the requirements of Lemma 5.

Detecting Swamps in Grids

Our swamp detection algorithm is based on the fact that each swamp-region contains at least one special node that we call a *seed*. We detect those seeds and then try to expand them into swamps.

Our definition of seeds is restricted to graphs that are represented by a 2D four-neighbor grid that may contain obstacles in some of the grid coordinates. All of our methods work for any other graph as well, except that, in those cases, more suspected nodes must be examined as possibilities for swamps. For simplicity of presentation, we will assume from now on that we are using only graphs that are represented by 2D four-neighbor grids.

We will start by defining a seed. Then, we will prove that each swamp-region contains at least one seed, and explain how we utilize this to detect swamps that have the properties needed for Lemma 5.

Definition 4. *In a four-neighbor 2-dimensional grid, a seed is a node $s \in V$ for which:*

1. *s is unblocked;*
2. *At least one of the nodes above or below s is blocked (or does not exist);*[3]
3. *At least one of the nodes to the right or left of s is blocked (or does not exist).*

Figure 4 displays a few seeds in a 2D grid.

Theorem 6. *Every swamp-region R contains at least one seed.*

The proof appears in the appendix.

We now present our algorithm for the detection of swamps. Our main goal is to assign as much of the grid as possible to swamp-regions, so that every subset of the regions composes a swamp-set. Better results will be obtained when we manage to cover more of the grid, as long as each region alone is not too large (so just considering the entire grid as one large swamp will give us very poor results).

[2]Note that search can also be reversed in directed graphs.

[3]If s is on the boundary of the graph then some of its neighbors do not exist.

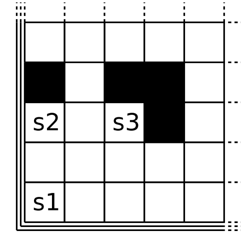


Figure 4: An example of seeds. $s1$, $s2$, and $s3$ are all seeds in this example.

The main idea of the algorithm is as follows. First, we detect all the seeds on the grid. Then, we iteratively extend each seed to a swamp-region, given the other swamp-regions that have already been found. We will now give more details about the algorithm and prove its correctness.

The Swamp Detection Algorithm

The pseudo-code of the swamp detection algorithm is described in Algorithm 1.[4] First, we initialize our swamp-regions to be the empty set and find all the seeds in the graph. Then, we try to extend each seed: first we check if it is a swamp-region by itself. If it is, we take the group of the seed plus all the nodes that it can reach in k moves (not including other swamp-regions), and try to trim it into a swamp (as explained later). We keep increasing k until we reach our size limit or until a few consecutive rounds of increasing k do not change our swamp size (notice that if we increase our radius by k and do not find a large swamp it does not mean that increasing by $k + 1$ will not find a larger swamp-region). We then return the largest swamp-region we have found so far.

Notice, that a swamp-set can be efficiently represented in memory. Each node in the graph needs just a few bits that tell to which swamp-region it belongs. This is very low cost (linear in the size of the graph), especially when considering currently available alternatives such as caching paths in the graph (where the number of paths is quadratic in its size, and therefore a large cache is needed to get significant improvements in performance).

We will now describe how we trim a group of nodes into a swamp that contains the seed (Algorithm 2). First, we find

[4]For simplicity of presentation, we used some functions without showing their implementation, if their implementation is trivial. Those functions are:

getReachable(seed, radius): returns nodes that can be reached from the seed in radius moves or fewer while counting $swamps^{t-1}$ as a swamp-set.

$findPath(v_1, v_2, S)$: searches and returns the shortest path between v_1 and v_2 under the assumption that S is a swamp-set.

We also assume that MAX is some predefined parameter set by the programmer.

Algorithm 1 The Swamp Detection Algorithm

```
procedure GROWSWAMPS(sizeLimit)
    swamps^0 = ∅
    seeds = detectSeeds()
    t = 1
    for all s ∈seeds do
        region = extendSeed(s, sizeLimit)
        if region not empty then
            swamps^t = swamps^{t−1} ∪ {region}
        t = t + 1

procedure EXTENDSEED(s, sizeLimit)
    radius = 0;
    size = 0;
    while radius < MAX AND size < sizeLimit do
        cluster = getReachable(seed, radius)
        current swamp = trimToSwamp(cluster, radius)
        if size(current swamp) > size then
            size = size(current swamp)
        radius = radius + 1
    return largest swamp found that had size less than sizeLimit
```

Algorithm 2 Trimming To Swamp Algorithm

```
procedure TRIMTOSWAMP(s, group)
    B = getBoundary(group)
    for all v1, v2 ∈ B do
        P1 = findPath(v1, v2, swamps^{t−1})
        P2 = findPath(v1, v2, swamps^t)
        if length(P2) > length(P1) then
            for all v_p2 ∈ P2 do
                if v_p2 ∈ group then
                    remove v_p2 from group
                    add v_p2 to B(group)
```

the boundary of the group including points that are also inside other existing swamp-regions. Then, we go over all pairs of points on the boundary, and search for the shortest path between them, twice. First, we search while ignoring the current group (but taking into account the other swamp-regions). Then, we search while counting our current group as a swamp-region as well. If the lengths of the paths differ, it means that the unification of this group with the rest of the swamp-set will not yield a valid swamp-set. We try to fix this by removing from the current group all nodes in the shortest path that passed through it, and then repeat the process.[5] We are left with a group of nodes that is a valid addition to the swamp-set. However, the trimmed-down group may no longer include the seed, or may no longer be a single connected component. To make sure we return a swamp that contains the seed, we only return remaining nodes in the group that are in the component of the seed.

Theorem 7. *After each stage* t *of the algorithm,* $swamps^t$ *is a swamp-set, and every subset of the regions in* $swamps^t$ *is also a swamp-set.*

Proof. The proof is by induction on the stages t of the algorithm. It is true for $t = 0$ and $t = 1$ from the definition of swamp-set and swamp-region. We will now prove that if we follow the algorithm and every subset of $swamps^{t-1}$ is a swamp-set then every subset of $swamps^t$ is also a swamp-set. Assume to the contrary that after stage t there is a subset of $swamps^t$ that is not a swamp-set. This means that the region $\mathcal{R}$ added at time t breaks the swamp-set conditions, when it is added to some subset of $swamps^{t-1}$, which we will denote as $regs^{t-1}$ (we know this subset to be a swamp-set from the induction assumption). Therefore there must exist v_1, v_2 such that searching from v_1 to v_2 while assuming $regs^{t-1} \cup \mathcal{R}$ is a swamp-set will not result in the shortest path. We know that $v_1, v_2 \notin \mathcal{R}$, otherwise $swamps^{t-1}$ would not be a swamp-set. Since there was a shortest path P_{v_1,v_2} from v_1 to v_2 under the assumption that $regs^{t-1}$ is a swamp-set, and it is blocked under the assumption that $regs^{t-1} \cup \{\mathcal{R}\}$ is a swamp-set, it means that the path must have passed through $\mathcal{R}$. Since v_1 and v_2 are not in $\mathcal{R}$, the path entered and left $\mathcal{R}$, so it passed through at least two nodes in $B(\mathcal{R})$. We will mark the first such node as v_{B1} and the last as v_{B2}. According to the algorithm we ran, there is a shortest path between v_{B1} and v_{B2} that is found under the assumption that $swamps^t$ is a swamp-setand therefore does not pass through $(reg^{t-1} \cup \{\mathcal{R}\}) \setminus \{reg(v_1), reg(c_2)\}$,[6] so we can replace the part of P_{v_1,v_2} between v_{B1} and v_{B2} with this path, thus getting a shortest path between v_1 and v_2 that can be found under the assumption that $regs^{t-1} \cup \{\mathcal{R}\}$ is a swamp-set, in contradiction to the claim. □

Note that Theorem 7 implies that our algorithm for detecting swamps is an anytime algorithm. At every stage, we have a swamp-set that is viable and we can use even partial results to improve path-finding. This suggests that instead of pre-processing the map we can detect swamps in between searches. Figure 5 illustrates the results of running our swamp detection algorithm.

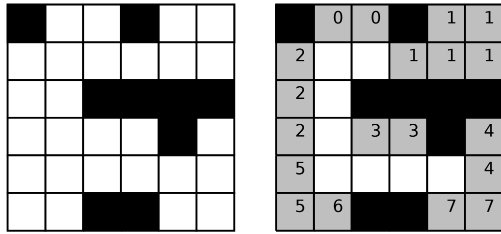
Experimental Results

We have run experiments on four-neighbor grids, where each node can be either blocked or free. Nodes were blocked at random with varying probabilities in each test, using different grid sizes. For each combination of grid size and probability of blocking a node, we generated 100 grids for the experiments. In each grid, nodes were independently blocked with equal probability.

We then proceeded to run our swamp detection algorithm on each generated grid as described in Algorithm 1. On the resulting processed map and swamp-set that was found in it, we ran 80,000 searches between pairs of points. Each search was repeated twice: once using regular A*, and once

[5] There may be several shortest paths that go through the group we are trimming to a swamp-region, and so there may be several ways to trim it. Some trimmings will not succeed, or may lead to the detection of different swamp-regions.

[6] We denote by $reg(v)$ the swamp region that contains v, or $\emptyset$ if v is not included in any swamp-region.



(a) The grid without the swamps. Obstacles are marked as black.

(b) The grid with swamps (marked in gray). The number indicates the swamp group to which a node belongs.

Figure 5: Example for the results of the swamp detection algorithm on a 6x6 grid, with 25 percent obstacles.

using the same implementation of A* but also using the additional information on the swamp-set we detected in the pre-processing stage.[7]

Our experiments demonstrated that using our detection and exploitation algorithm results in a significant saving in the search cost, in terms of expanded nodes. Figure 6 compares the cost searching with and without swamps on different grid sizes and with a different probability of generating obstacles. The figure also shows the average path length during searches. Note that the number of nodes expanded in our approach is significantly lower than the number of expanded nodes during a regular activation of A*. The saving becomes more and more pronounced in larger grid sizes, where A* expands many more nodes than are strictly needed for the path. The density of obstacles is also a factor in the efficiency of the method. As the number of obstacles rises, so does our algorithm's savings.

Figure 7 shows the results of experiments that were run on a 60x60 grid that had a 30% obstacle density. A path is considered x-efficient if the number of nodes expanded to find it is x times more than the path length. The figure presents the percentage of x-efficient paths, and it is clear that utilizing information about swamps increases the efficiency of search for both hard and easy paths.

The increase in efficiency when using swamps on larger grid sizes and a higher obstacle density can be explained by the fact that our pre-processing algorithm manages to locate more swamp-regions in the grid. Figure 8 shows the expected amount of squares that are categorized as swamp

[7]Only pairs of points that were connected were used in the experiments. The reason for choosing only nodes that have a path between them is that when A* searches between two nodes, s and t, that do not have any path between them, it will expand all the nodes in the connected component of s. This is an unfair advantage for our algorithm that uses swamps, as the connected component of s in the pruned graph that is received after removing swamps is significantly smaller than the connected component in the full graph. Therefore, A* that uses swamps will in these cases have an obvious advantage. Counting such pairs would only improve the performance of our algorithm when compared to A*.

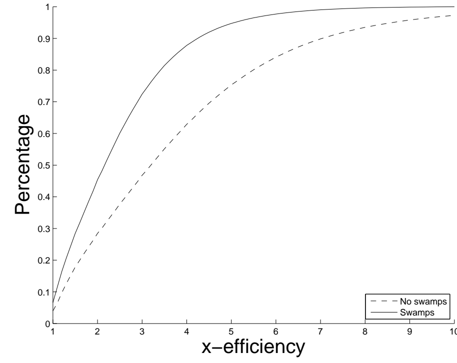


Figure 7: Percent of x-efficient paths with and without swamps—expected value

nodes vs. the number of blocked nodes and free (non-swamp) nodes in the graph with different grid sizes and obstacle percentages. With a higher percentage of obstacles, the number of swamp nodes greatly exceeds the number of free nodes, and the search runs on a much smaller graph.

The increase in savings on larger grids can be explained by the fact that in larger grids the paths tends to be longer, and may therefore pass near a greater number of swamp-regions. Also, each single swamp-region contributes to the efficiency of many paths.

Finally, Figure 9 presents a graph that shows the benefits of reversing the search direction if the search starts outside of a swamp and proceeds into it. The number of expanded nodes is decreased when searching into a swamp-region when compared to the reversed search.

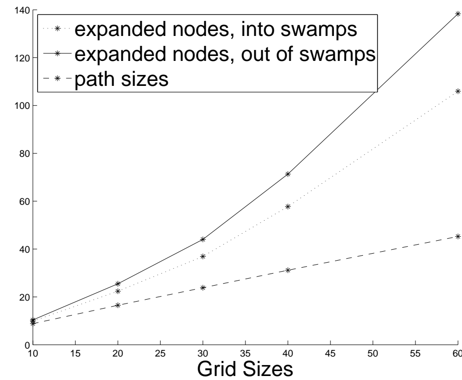
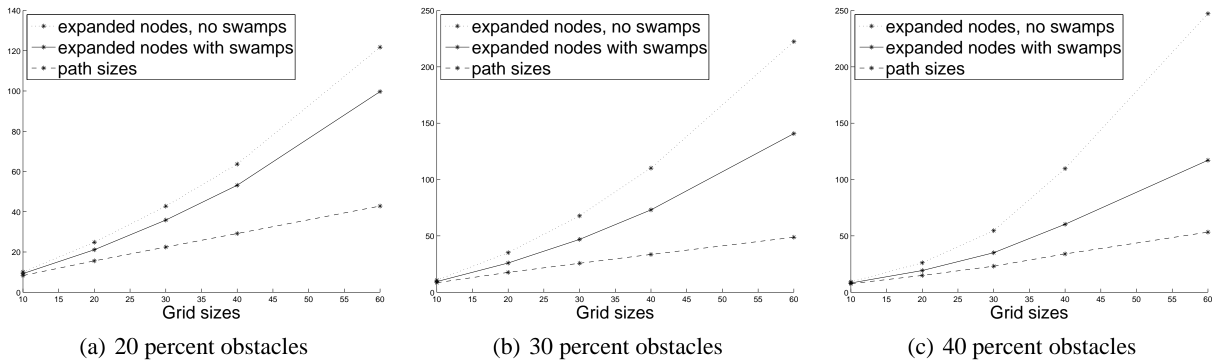


Figure 9: The effects of reversing search that starts outside a swamp-region (20% obstacle density)

Discussion and Future Work

In this paper, we introduced swamps—groups of nodes in a graph that can hinder the search process. We formally defined swamp-sets and swamp-regions, and presented an algorithm for using swamp-sets to reduce search cost while still detecting optimal paths. We then presented an anytime

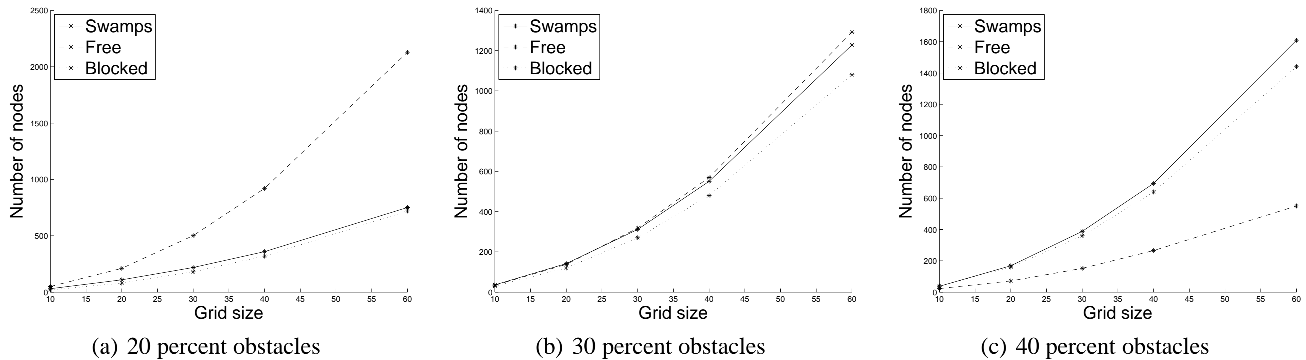


(a) 20 percent obstacles

(b) 30 percent obstacles

(c) 40 percent obstacles

Figure 6: Percentage of savings using swamps with different grid sizes and obstacle percentages



(a) 20 percent obstacles

(b) 30 percent obstacles

(c) 40 percent obstacles

Figure 8: Number of free, blocked, and swamp nodes with different grid sizes and obstacle percentages

algorithm that detects swamps in two-dimensional four-neighbor grids (although our algorithm is easily adaptable to any non-hidden graph). We formally proved that using this algorithm returns a swamp-set that satisfies the properties needed for the exploitation algorithm to work correctly. We then demonstrated with experiments on random grids that the above algorithm greatly reduces search cost, i.e., the number of nodes expanded during the search. Our algorithm requires very little memory—only a few bits per node on the graph in order to assign that node to some swamp-region.

It still remains to test our approach on different types of graphs with various search algorithms. We also believe that the detection of swamp-sets can be greatly improved using other methods to grow swamps. Since our approach can be combined with other algorithms and heuristics to improve search, it would be interesting to attempt to boost other efficient search methods with it. We also believe that it may be of value to use other, more complex sets of swamps that perhaps utilize information about different swamp-sets that cannot be directly unified using our current approach.

References

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 1(1):7–28.

Hart, P.; Nilsson, N.; and Raphael, B. July 1968. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on* 4(2):100–107.

Koenig, S., and Likhachev, M. 2002. D* lite. In *AAAI'02*, 476–483.

Koenig, S., and Likhachev, M. 2006. Real-time adaptive A*. In *AAMAS'06*, 281–288. New York, NY, USA: ACM.

Koenig, S.; Likhachev, M.; and Furcy, D. 2004. Lifelong planning A*. *Artif. Intell.* 155(1-2):93–146.

Koenig, S. 2004. A comparison of fast search methods for real-time situated agents. In *AAMAS*, 864–871. IEEE Computer Society.

Korf, R. E. 1985. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intell.* 27(1):97–109.

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

Sturtevant, N. R., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *AAAI*, 1392–1397.

Sturtevant, N. R. 2007. Memory-efficient abstractions for pathfinding. In *AIIDE*, 31–36.

Proof of Theorem 6

Theorem. *Every swamp-region R contains at least 1 seed.*

Definition 5. *We say that a shortest path P between nodes v_1, v_2 is Manhattan if its length is exactly the Manhattan distance between v_1 and v_2.*

Note that any Manhattan path can only consist of moves in 2 perpendicular directions (e.g., up and to the right). If it consists of more than two then it takes more steps than the Manhattan distance between the nodes because it goes in two opposite directions (somewhere along the path), and both these opposite moves cancel out when considering the change in coordinates along the path.

Lemma 8. *In a 2D four-neighbor grid, if a Manhattan path P passes through a connected component $\mathcal{R}$, and both steps of entering and exiting the swamp-region are taken in the same direction, then $\mathcal{R}$ is not a swamp-region.*

Proof of lemma. Let us arbitrarily name the direction of entrance and exit used by path P as up, and assume w.l.o.g. that the path P only proceeds in steps that are either up or to the right (otherwise it is not a Manhattan path). Figure 10 illustrates a path taken through the swamp-region.

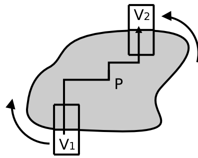
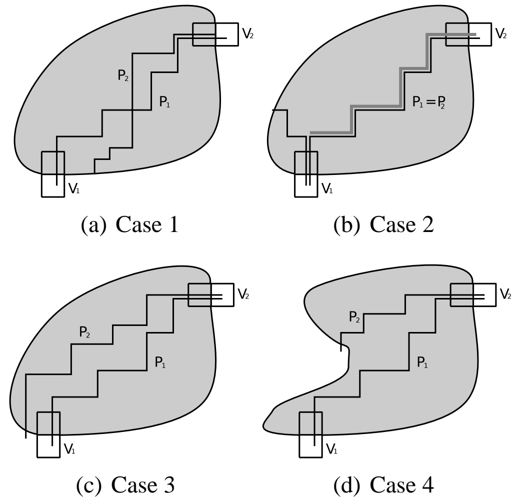


Figure 10: A path cutting through the swamp-region that exits and enters in the same direction

Let v_1, v_2 be the two endpoints of P. Because P is Manhattan, it is a shortest path between its endpoints, both of which are outside $\mathcal{R}$. It is therefore sufficient to show that no other shortest path can connect these two points without passing through $\mathcal{R}$. Since $\mathcal{R}$ is a connected component, in order to go from v_1 to v_2, a path must go either clockwise around $\mathcal{R}$, or counter-clockwise. Any path that goes clockwise will have to start at v_1 and visit a node on the graph that is to the left of v_1. It therefore moves left at some point, and must move to the right later (because v_2 is above v_1 and to the right). Therefore a clockwise path is in fact longer. A similar reasoning applies to a counter-clockwise path, that must visit a point that is to the right of v_2 and then proceed to the left towards v_2. Therefore the only optimal paths between v_1 and v_2 must pass through $\mathcal{R}$. □

Proof of theorem. Let $\mathcal{R}$ be some swamp-region, and let us assume to the contrary, that $\mathcal{R}$ does not contain any seeds. Let v be some unblocked node inside $\mathcal{R}$. Since there are no seeds in $\mathcal{R}$, it must be possible to proceed either left or down from v (otherwise, both are blocked and we have a seed). After taking 1 such step it must always be possible to take another, and go on in this manner until eventually exiting $\mathcal{R}$. Without loss of generality, we assume that the last step out of $\mathcal{R}$ is a step down, into node u. Therefore, when walking up from u the region $\mathcal{R}$ is entered. Let v_1 be the node furthest to the left that is unblocked and for which a step up takes us into $\mathcal{R}$ (there must exist at least one such node—u). From v_1 let us take a path P_1 that goes up whenever the node above is unblocked, and right when the node above is blocked. Since P_1 is a Manhattan path, it cannot end by exiting $\mathcal{R}$ in a step that goes up (otherwise, according to Lemma 8, $\mathcal{R}$ is not a swamp-region). Therefore, the path P_1 ends in a right step that reaches some node v_2 outside $\mathcal{R}$. v_2 therefore has a left entrance into $\mathcal{R}$. Now, let P_2 be a path that starts at v_2 and proceeds left whenever possible, otherwise it will proceed down. This path eventually leaves the swamp (again, it exists because $\mathcal{R}$ has no seeds). It is impossible that this path leaves the swamp-region in a left step because then we would reach a contradiction according to Lemma 8. There are now 4 possibilities (each of which will lead us to a contradiction) as depicted in Figure 11.



(a) Case 1 (b) Case 2

(c) Case 3 (d) Case 4

Figure 11: Various cases in the proof of Theorem 6

1. The path P_2 intersects P_1 and passes below it. This is only possible if P_2 goes down and meets P_1 at some point, which contradicts the way path P_1 was constructed—always preferring to go up whenever possible.
2. The path P_2 is identical to P_1. This implies that every point above P_1 is blocked (as it chooses to go up whenever possible) and so is every point to its left (because of the way P_2 was constructed). This implies that $\mathcal{R}$ has a seed.
3. The path P_2 exits at some coordinate to the left of v_1. This is impossible because v_1 was selected to be the node furthest to the left that has an entrance in an upward step.
4. The path P_2 exits above and to the right of v_1. In this case, P_2 is a Manhattan path and has no alternative outside the region $\mathcal{R}$, which is therefore not a swamp-set. Since $\mathcal{R}$ is a connected component, any alternative to P_2 must either go clockwise around $\mathcal{R}$ or counter-clockwise. If it goes clockwise, it must pass above node v_2 and then proceed down towards it. The path is therefore not optimal. If it proceeds counter-clockwise, it must proceed below v_1 and then go up again. In any case, this path is not optimal. P_2, however, is optimal, and we reach a contradiction.

We have therefore reached a contradiction in every case. □