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, Israel
{nirp, avivz, jeff}@cs.huji.ac.il

ABSTRACT

In various 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 known graphs. We propose a method that relies on identifying areas that tend to be searched needlessly (areas we call *swamps*), and exploits this knowledge to improve search. The method requires a relatively small amount of memory, 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 swamps.

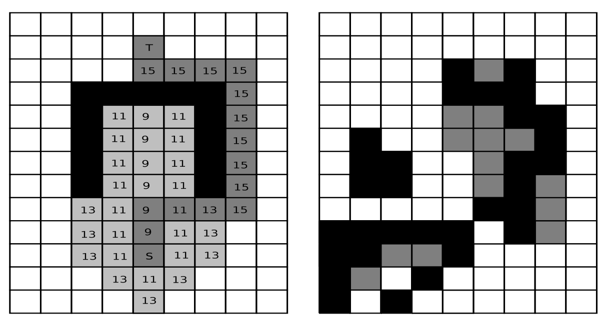
1. INTRODUCTION

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 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(a); in this example, algorithms such as A* [2] (with a good heuristic) can search very efficiently in some areas of the map, while being very inefficient in other areas. Figure 1(a) 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(b) shows a less obvious example for nodes with a similar property—all nodes that are marked in gray in this example never have to be considered in the search for a shortest path, unless we start or end our search at them.

Our approach will be to automatically identify areas such as the cup, which we will call *swamps*, and efficiently store information about them in the graph (not using 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 swamps 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 swamps to reflect minor changes in the environment.



(a) 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.

(b) All nodes that are marked in gray can be ignored during search in this eight-way grid, unless we search into or out of them

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

*Aviv Zohar is also affiliated with Microsoft Israel R&D center, Herzlia, Israel. All authors partially supported by Israel Science Foundation grant #898/05.

Using Swamps to Improve Optimal Pathfinding, Nir Pochter, Aviv Zohar and Jeffrey S. Rosenschein,

We then explain how to exploit information about swamps during the search so as to obtain shortest paths while expanding fewer nodes. Next, we present an algorithm to detect swamps in general graphs, and prove its correctness. We then present experimental results that support the claim that using our algorithms on four and eight-neighbor grids significantly reduces search costs. We conclude by discussing future work.

1.1 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* [2] and IDA* [7] are widely used, where A* is usually faster but can consume more memory than IDA*.

Several methods, such as [9], [1], and [10], 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 on 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* [8, 3] and RTAA [5] 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* [6] and D* lite [4] reuse previous search information when the environment is dynamic; here, 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 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 to many others; it does not compete with them. Our algorithm just adds a pre-processing stage that should be executed once per graph.

2. SWAMPS

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 $\mathcal{S}$ in an undirected graph $G = (V, E)$ is a group of nodes $\mathcal{S} \subseteq V$ such that any 2 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 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.

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

[1]A slightly more restrictive alternative is to define a swamp 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 swamps and is thus less useful in practice.

The next example illustrates the definition of a swamp.

EXAMPLE 1. *Figure 1 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 $\{s_1, \ldots, s_4\}$.*

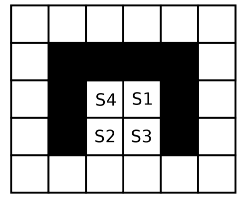


Figure 1: An example of a swamp-region. Nodes filled in black are obstacles. Nodes $\{s_1, s_2, s_3, s_4\}$ form a swamp-region.

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

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

2.1 Additional Properties of Swamps

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 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. This will later give us a good procedure for checking if a given set of nodes is a swamp, 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.*

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 passes 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 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.

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

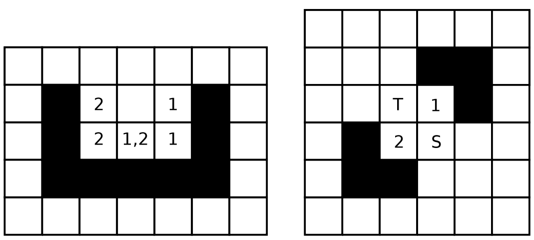
PROOF. According to Lemma 1, it is enough to show that any two points on the boundary of $\mathcal{R}$ have a connecting shortest path that does not pass through $\mathcal{R}$. We know this is true, because by definition $B(\mathcal{R})$ consists only of obstacles or nodes that do not belong to $\mathcal{S}$. Therefore, because $\mathcal{S}$ is a swamp, 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 $\mathcal{S}$. □

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 swamps 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 would compose a smaller swamp. This is not the case. In fact, even the intersection of two known swamps (which is therefore contained in both) is not necessarily a swamp.

LEMMA 3. *The intersection of two swamps, $\mathcal{S}_1$ and $\mathcal{S}_2$, is not necessarily a swamp.*

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



(a) An example that shows that an intersection of swamps is not necessarily a swamp.

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

Figure 2: Figures used in the proofs of Lemmas 3 and 4

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

LEMMA 4. *The unification of two swamps, $\mathcal{S}_1$ and $\mathcal{S}_2$, may not be a swamp.*

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

3. USING SWAMPS TO DECREASE SEARCH COSTS

A naive approach to using a swamp to lower search costs is to consider them as blocked whenever a search between two nodes from outside the swamp is performed. The search is then performed on an effectively smaller graph, and could be expected to open fewer nodes. By the definition of swamps, 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.

We will try to increase the benefits we get from swamps by using a swamp that is completely partitioned into different swamp-regions. For this purpose, we add the following definition:

DEFINITION 4. *A swamp-collection $\mathcal{C}$ is a set of swamp-regions, any subset of which forms a swamp together.*

$$\mathcal{C} = \{\mathcal{R}_1, \dots \mathcal{R}_k\}$$

Notice that while every swamp can be broken down into connected components which together form a swamp-collection, a swamp-collection may also be composed of regions that are more finely granulated. That is, a connected component that is part of a swamp can be broken down into smaller components that are each a swamp-region.

As we later show, we can find large swamp-collections in our graphs. The advantage of using swamp-collections is that when we search between two nodes in the graph, we can consider any swamp-region that they do not belong to as blocked, and thus achieve significant savings on searches between swamp nodes as well.

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 $\mathcal{C} = \{\mathcal{R}_1, \dots, \mathcal{R}_k\}$ be the full swamp-collection that was found in the graph.
3. Let $\mathcal{R}' \in \mathcal{C}$ be the swamp-region that v_1 belongs to, or $\emptyset$ if v_1 does not belong to any swamp-region.
4. Let $\mathcal{R}'' \in \mathcal{C}$ 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

$$\left(V \setminus \bigcup_{i=1}^{k} \mathcal{R}_i\right) \cup \mathcal{R}' \cup \mathcal{R}''$$

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. Because $\mathcal{C}$ is a swamp-collection, we know that the nodes in the regions $\mathcal{C} \setminus \{\mathcal{R}', \mathcal{R}''\}$ form a swamp together, and therefore any search that ignores those nodes can still produce an optimal path. □

In the next section, we will show how to find a swamp-collection, i.e., a set of swamp-regions that will satisfy the requirements of Lemma 5.

4. DETECTING SWAMPS IN GRIDS

Our swamp detection algorithm requires going over nodes and checking if they can extended into swamps. Running this test on every node in the graph, however, can be a very difficult task. To help the detection phase go faster, we make use of the fact that certain types of graphs have a special type of node, which we call *seeds*, such that every swamp-region must contain at least one node of this type. This will later imply that it is sufficient to run the above test only on those nodes.

DEFINITION 5. *In a graph $G = (V, E)$, a seed is a node $s \in V$ which is surrounded by a certain structure of nodes, such that every swamp-region in G must contain at least one node with this structure.*

The definition means that we know that each swamp-region contains at least one node of the group of all seeds, so we only need to try and test the seeds. Generally, we can always use the group of all nodes in a graph as the group of seeds, as every swamp-region must contain at least one node. This, however, does not help much, as we would still be trying to extend all nodes into swamp-regions. We will now show that four-neighbor and eight-neighbor seeds, however, constitute a relatively small group out of all the nodes in the graph, and are easy to detect.

We claim that on a four-neighbor grid, a seed is a node that has the following structure:
1. s is unblocked;
2. At least one of the nodes above or below s is blocked (or does not exist);[2]
3. At least one of the nodes to the right or left of s is blocked (or does not exist).

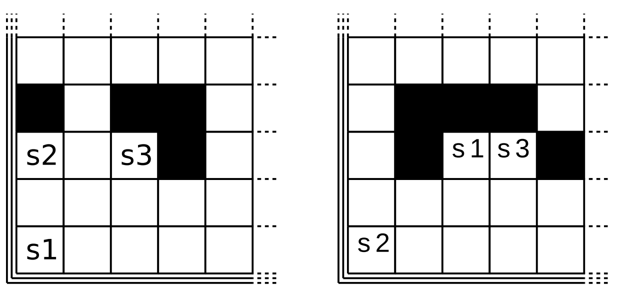
THEOREM 6. *Every swamp-region R in a four-neighbor grid contains at least 1 seed (a node with the structure defined above).*

The proof appears in the appendix.

Seeds in eight-neighbor grids[3] have a more complicated form, so we will introduce them in a less formal way for ease of presentation. We claim that a seed in an eight-neighbor grid is a node that is unblocked, and is "trapped" in an L-shaped form (similar to the L-shaped form in Tetris), which can be mirrored or rotated. Examples of seeds in eight-neighbor grids can be found in Figure 3(b). While we do not have a formal proof that this indeed describes a seed in eight-neighbor grids, we ran many experiments on different sizes of grids and different obstacle density, and in all of them every swamp-region contained at least one node with this structure. We therefore conjecture the following:

CONJECTURE 1. *Every swamp-region in an eight-neighbor grid must contain at least one node with the structure defined above.*

Figure 3(a) displays a few seeds in a 2D grid.



(a) An example of seeds in a four-neighbor grid. $s1$, $s2$, and $s3$ are all seeds in this example.

(b) An example of seeds in an eight-neighbor grid. $s1$ and $s2$ are seeds in this example; $s3$ is not a seed.

We now present our algorithm for the detection of swamp-collections. 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. 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]If s is on the boundary of the graph then some of its neighbors do not exist.

[3]In this paper we assume that in eight-neighbor grids a diagonal move costs $\sqrt{2}$.

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, while preserving the properties of the swamp-collection that has been found so far. We now give more details about the algorithm, and prove its correctness.

4.1 The Swamp Detection Algorithm

The pseudo-code of the swamp detection algorithm is described in Algorithm 1.[4] First, we initialize our swamp-collection 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.

Algorithm 1 The Swamp Detection Algorithm

procedure GROWSWAMPS(sizeLimit)
  $collection^0 = \emptyset$
  seeds = detectSeeds()
  $t = 1$
  for all s ∈seeds **do**
    $region$ = extendSeed(s, sizeLimit)
    if $region$ not empty **then**
      $collection^t = collection^{t-1} \cup \{region\}$
    $t = t + 1$

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

Notice, that a swamp-collection 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 the boundary of the group including points that are also inside other existing

[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.
$findPath(v_1, v_2, S)$: searches and returns the shortest path between v_1 and v_2 under the assumption that S is a swamp.
We also assume that MAX is some predefined parameter set by the programmer.

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-collection will not yield a valid swamp-collection. 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-collection. 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.

Algorithm 2 Trimming To Swamp Algorithm

procedure TRIMTOSWAMP($s, group$)
  $B = getBoundary(group)$
  for all $v_1, v_2 \in B$ **do**
    $P1$ = findPath(v_1, v_2, $collection^{t-1}$)
    $P2$ = findPath(v_1, v_2, $collection^{t}$)
    if $length(P2) > length(P1)$ **then**
      for all $v_{p2} \in P2$ **do**
        if $v_{p2} \in group$ **then**
          remove v_{p2} from $group$
          add v_{p2} to $B(group)$

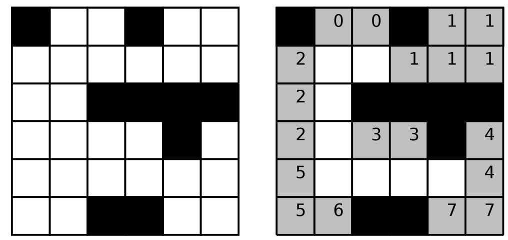
THEOREM 7. *After each stage t of the algorithm, $collection^t$ is a swamp-collection, and thus every subset of the regions in $collection^t$ is also a swamp.*

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 and swamp-region. We will now prove that if we follow the algorithm and every subset of $collection^{t-1}$ is a swamp-collection then every subset of $collection^t$ is also a swamp-collection. Assume to the contrary that after stage t there is a subset of $collection^t$ that is not a swamp. This means that the region $\mathcal{R}$ added at time t breaks the swamp-collection conditions, when it is added to some subset of $collection^{t-1}$, which we will denote as $regs^{t-1}$ (we know this subset to be a swamp 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 will not result in the shortest path. We know that $v_1, v_2 \notin \mathcal{R}$, otherwise $collection^{t-1}$ would not be a swamp-collection. 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, and it is blocked under the assumption that $regs^{t-1} \cup \{\mathcal{R}\}$ is a swamp, 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 $collection^t$ is a swamp and 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, in contradiction to the assumption. □

[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.

Note that Theorem 7 implies that our algorithm for detecting swamps is an anytime algorithm. At every stage, we have a swamp 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 3 illustrates the results of running our swamp detection algorithm.



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

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

Figure 3: Example for the results of the swamp detection algorithm on a 6x6 grid, with 25 percent obstacles.

5. EXPERIMENTAL RESULTS

We have run experiments on four and eight-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 random grids for the experiments. In each grid, nodes were independently blocked with equal probability.[7] We then ran our swamp detection algorithms, and various searches,[8] with and without the swamps, to measure performance. We will present here two different types of measurement: the time it takes to perform the search, and the number of nodes expanded. The reason for presenting both is that smaller node expansion using swamps does not necessarily mean that search will be faster, as the exploitation algorithm incurs the overhead of checking whether a new node is a swamp, and whether it should be expanded or not.

Implementation details can make a big difference in the efficiency of exploiting swamps—and as we tried to have an implementation that is as general as possible, we believe that even better results can be achieved for specific tasks. Our implementation was

[7]Since 8-way graphs are much more connected, as they contain more edges, we used higher obstacle densities for the 8-way grids—otherwise a search in these graphs is too easy.

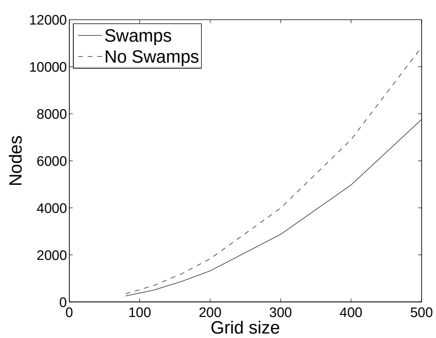
[8]Only pairs of points that have some path between them were used in the experiments. The reason for choosing only nodes that are connected in this way is that when A* searches between two nodes, s and t, that do not have any path between them, it will expand all the nodes in the connected component 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*.

written in Java; the experiments that were used to measure run-time performance were carried out on an Intel Pentium 4, 2.4GHZ machine, with 500MB of RAM.

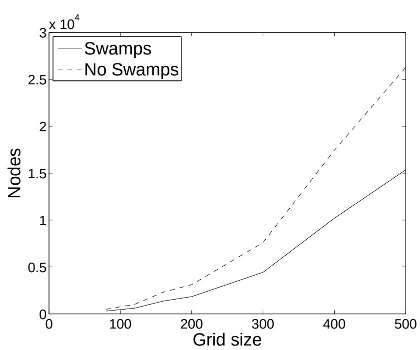
5.1 Node Expansion Measurements

We ran our swamp detection algorithm on each generated grid as described in Algorithm 1. We ran 1,000 searches between pairs of points. Each search was repeated twice: once using regular A*, and once using the same implementation of A* but also using the additional information on the swamp that was detected in the pre-processing stage.

Our experiments demonstrated that using our detection and exploitation algorithm results in a significant saving in the search cost, in terms of expanded nodes. Figures 4 and 5 compare the costs of searching with and without swamps on different grid sizes and with a different probability of generating obstacles on four and eight neighbor grids, respectively. The figures also show the average path length (in number of nodes) 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.



(a) 30 percent obstacles — 4-way grid

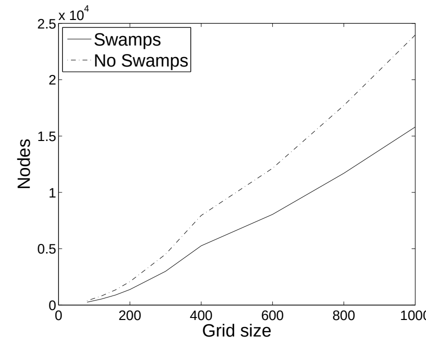


(b) 40 percent obstacles — 4-way grid

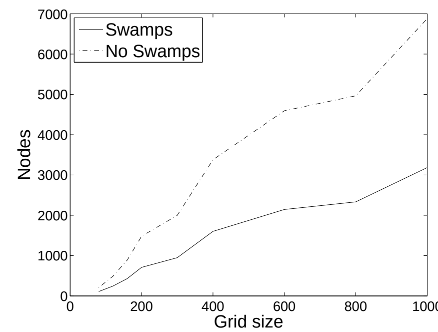
Figure 4: Expanded nodes (swamps, no swamps) and path size, 4-way grid

5.2 Time Measurements

In addition to the number of expanded nodes, we also measured the time it took to detect swamps, and the time it took to execute the searches with and without swamps. It is important to remember



(a) 50 percent obstacles — 8-way grid



(b) 60 percent obstacles — 8-way grid

Figure 5: Expanded nodes (swamps, no swamps) and path size, 8-way grid

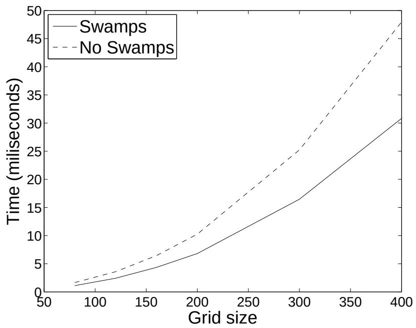
that time measurement is a risky metric. First, time measurement depends on operating system and language implementations. Second, running time can be dependent on the architecture used in the measured program, and its implementation details. However, we still feel that these measurements are important as a rough demonstration of the potential benefit of our algorithm.

The settings of our first measurement were similar to the settings of the previous measurement of expanded nodes—we ran our pre-processing algorithm to find the swamps, and then ran the search twice between 1,000 pairs of points, once with and once without utilizing swamps, measuring the run-times. Figures 6 and 7 display the comparison of search time for four-neighbor and eight-neighbor grids. The figures shows that the saving in the number of nodes expanded translated to a saving in the search time.

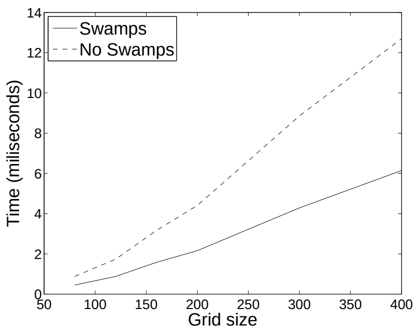
Another measurement which might be more interesting is the number of searches, on average, that it takes to make up for the time it took to perform the swamp pre-processing. This data for four and eight-neighbor grids is shown in Figures 8 and 9. The figures demonstrate that the state pre-processing cost is returned after a few hundred searches, and that this number decreases as the size of the grid increases.

6. 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 swamps, swamp-regions, and swamp-collections, and presented an algorithm for using swamp-collections to reduce search cost while still detecting optimal paths. We then presented an anytime algorithm that detects swamps in two-dimensional four-neighbor and eight-neighbor grids (although our algorithm is easily adaptable to any

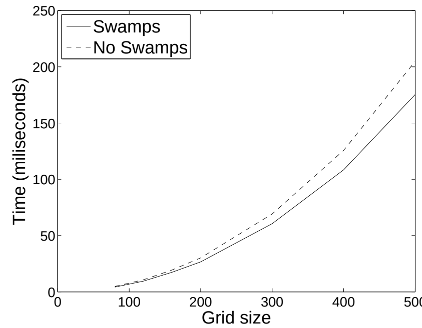


(a) 50 percent obstacles

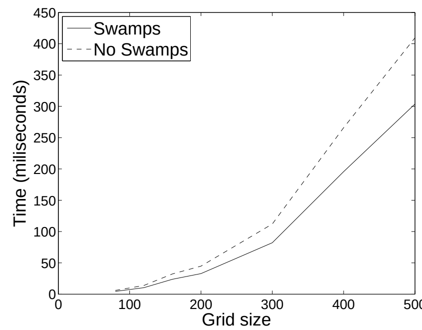


(b) 60 percent obstacles

Figure 6: Time it took to execute searches with and without swamps, 8-way grids



(a) 30 percent obstacles



(b) 40 percent obstacles

Figure 7: Time it took to execute searches with and without swamps, 4-way grids

non-hidden graph). We formally proved that this algorithm returns a swamp-collection that satisfies some extra 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, and the time it took to perform 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. Since our approach can be combined with other algorithms and heuristics to improve search, it would be interesting to attempt to boost the efficiency of other search methods with it.

7. REFERENCES

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

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

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

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

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

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

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

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

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

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

APPENDIX

A. PROOF OF THEOREM 6

Theorem 6: *Every swamp-region R in a four-neighbor grid contains at least 1 seed.*

DEFINITION 6. *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 enter-*

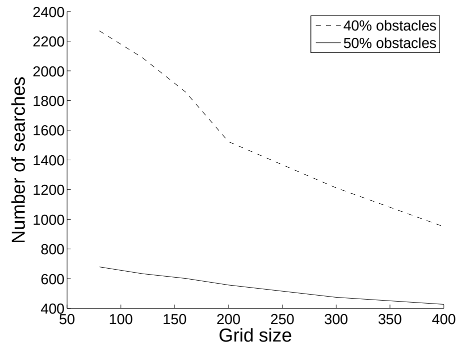


Figure 8: The average number of searches needed to make up for the time it cost to pre-process the graph (8-way grid))

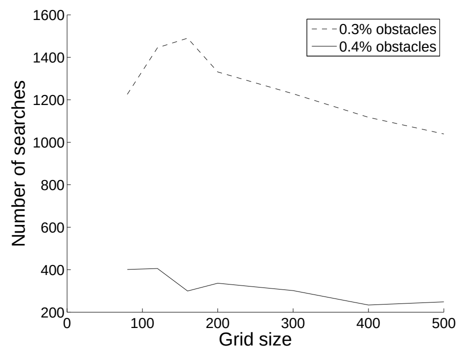


Figure 9: The average number of searches needed to make up for the time it cost to pre-process the graph (4-way grid)

ing and exiting the swamp-region are taken in the same direction, then $\mathcal{R}$ is not a swamp-region.

PROOF 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.

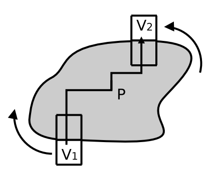
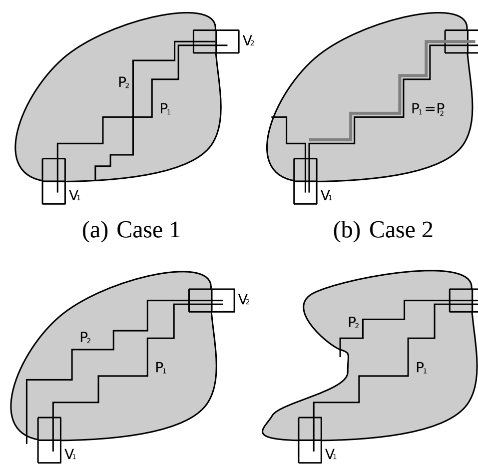


Figure 10: A path cutting through the swamp-region that exits and enters in the same direction

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. 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. $\square$