Search Space Reduction Using Swamp Hierarchies

Nir Pochter
School of Engineering and Computer Science
The Hebrew University, Jerusalem, Israel
nirp@cs.huji.ac.il

Aviv Zohar
School of Engineering and Computer Science
The Hebrew University, Jerusalem, Israel
and Microsoft Israel R&D Center Herzlia, Israel
avivz@cs.huji.ac.il

Jeffrey S. Rosenschein
School of Engineering and Computer Science
The Hebrew University, Jerusalem, Israel
jeff@cs.huji.ac.il

Ariel Felner
Information Systems Engineering
Ben-Gurion University, Be'er-Sheva, Israel
felner@bgu.ac.il

Abstract

In various domains, such as computer games, robotics, and transportation networks, shortest paths may need to be found quickly. Search time can be significantly reduced if it is known which parts of the graph include "swamps"—areas that cannot lie on the only available shortest path, and can thus safely be pruned during search. We introduce an algorithm for detecting hierarchies of swamps, and exploiting them. Experiments support our claims of improved efficiency, showing significant reduction in search time.

Introduction

A common direction in heuristic search is to develop techniques for very large combinatorial domains (e.g., permutation puzzles) where the state space is defined only implicitly, due to its exponential size. However, there are many domains, such as map-based searches (common in GPS navigation, computer games, and robotics) where the entire state-space is given explicitly. Optimal paths for such domains can be found relatively quickly with simple heuristics, especially when compared to the time it takes to explore exponentially large combinatorial problems. Relative quickness, however, might still not be fast enough in certain real-time applications, where further improvement towards high-speed performance is especially valued.

We present an approach that relies on preprocessing techniques that can dramatically reduce search costs, and do not compromise search optimality (A preliminary version appeared in (Pochter, Zohar, and Rosenschein 2009)). Our preprocessing determines the location of *swamps*, namely areas that can always be safely pruned, as long as they do not contain the start or end state. This approach is particularly useful when maps are known in advance and are used for multiple searches.

To understand the intuition behind swamps, think of an agent traversing a maze. A certain corridor in the maze may be the long path to the target or even a dead end, and thus may be useless for constructing short paths. A search algorithm may still look inside this corridor, especially if the heuristic indicates that this corridor is in the general direction of the target, and should be explored before other options. Only when the corridor is further explored will the algorithm learn that it does not lead to the target quickly. We automatically identify such areas in the graph during the preprocessing stage, and allow the search algorithm to explore them only under very specific circumstances: when the search originates or terminates within them.

We build upon this basic intuition, and present techniques that identify more complex structures that can be partially pruned, improving search even more. Experimental analysis of our approach in various domains reveals a drastic decrease in search costs when compared to search algorithms without this preprocessing.

Related Work A common technique for speeding up heuristic search is through preprocessing techniques. The work that is perhaps closest to ours is the "dead-end heuristic" introduced by Björnsson and Halldórsson (2006). They use a preprocessing phase to identify areas that are dead-ends, and create an abstract graph whose nodes are these areas. Initially, the search is performed on the abstracted graph. The areas that were not visited during the search on the abstracted graph are then ignored when the search is performed in the original search space. In addition to identifying dead-ends, our approach also identifies (and prunes, when possible) areas when there is an alternative shortest path that avoids them. We also do not require any additional searches in abstract spaces, and we utilize a hierarchical approach that saves significantly more time during the search.

Geisberger et al. (2008) used contraction to replace vertices through which few shortest paths pass, with shortcuts. They applied their technique to the Dijkstra algorithm using GPS data.

There is much research on efficient search in explicit state-spaces, using techniques that compromise solution quality using different graph abstraction techniques (Sturtevant 2007; Botea, Müller, and Schaeffer 2004; Sturtevant and Buro 2005; Demyen and Buro 2006). The basic idea of these techniques is to run a preprocessing phase, during which the original graph is abstracted, sometimes to multiple levels. The search is performed in an abstract graph, and mapped back to the original graph where it is then refined.

The resulting path, however, is not guaranteed optimal.

Other preprocessing techniques encode memory-based heuristics in the form of lookup tables. *Pattern databases* (PDBs) (Culberson and Schaeffer 1998), and their enhancements, have recently been explored as a powerful method for automatically building admissible memory-based heuristics based on abstractions of the domain. PDBs are usually goal-specific, and are usually designed and used for implicit exponential domains.

Another class of memory-based heuristics, *true distance heuristics* (TDHs), was introduced for explicit state spaces (Sturtevant et al. 2009; Felner et al. 2009). Ideally, the *all-pairs-shortest path* matrix can be precomputed and stored. Due to time and memory constraints, TDHs store shortest distances between a small number of selected pairs of states in the state space, using them to calculate an admissible heuristic. This method is naturally more precise when more memory is available for the cached distances. Our own approach achieves equivalent results with less memory.

All these preprocessing methods are intended to speed up the run-time search. It is commonly assumed that the time requirements of the preprocessing phase are of lesser importance, as it is performed only once but can then can be amortized over the solving of many problem instances.

Our own technique does not introduce a new heuristic, but rather prunes the graph; in principle, it can be used with any search algorithm, heuristic, or approach that uncovers new heuristics (including PDBs and TDHs).

Swamps

In this section we present the notion of swamps (the underlying concept used in our approach), starting with the basic definition, and then develop the more complex notions of *modular-swamps* and *swamp-hierarchies*.

A swamp is defined as a subset of the states in the search graph such that any shortest path that goes through the swamp (and does not start or end in it), has an equally short alternative that does not pass through it.[1] We define this notion more formally below.

Definition 1. *A* swamp $\mathcal{S}$ *in a graph* $G = (V, E)$ *is a set of states* $\mathcal{S} \subseteq V$ *such that for each* $v_1, v_2 \in V \setminus \mathcal{S}$*, there exists a shortest path* $P_{1,2}$ *that connects* v_1 *and* v_2 *and traverses only nodes in* $V \setminus \mathcal{S}$*.*

Note that in general, a swamp does not have to be a connected subset of states. We will use the term *contiguous-swamp* to specify a *connected* subset of states that is a swamp. We will often denote it in this paper by $\mathcal{C}$.

Example 1. *Figure 1 shows two contiguous-swamps,* $\mathcal{C}_1$ *which is composed of state 1, and* $\mathcal{C}_2$ *which is composed of states 5 and 6. Additionally, the set of states* $S = \{1, 5, 6\}$ *forms a swamp: all shortest paths between the remaining states in the graph do not pass through* S*.*

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

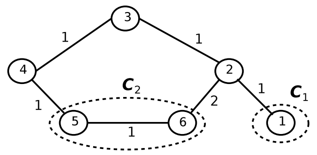


Figure 1: An example of two contiguous-swamps. Note that their unification is also a disconnected swamp.

Definition 2. *The* external boundary *of a set of states* T*,* $B(T)$*, is the set of nodes outside the set that have neighboring nodes in the set.*

Note that this definition applies to every set of states, not just connected sets.

The next lemma states that it is enough to check only paths between points on the boundary of a swamp in order to ensure that it is indeed a swamp. This gives us an efficient and local procedure for checking if a given set of nodes is a swamp by iterating over all pairs of nodes on the boundary of the candidate set.

Lemma 1. *Let* $\mathcal{S}$ *be a set of vertices in* V*. If for any two vertices 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.*

Proofs throughout are omitted due to lack of space.

The exploitation of swamps during search occurs as follows: when a search is performed between two states, both of which are outside of a swamp $\mathcal{S}$, all nodes belonging to $\mathcal{S}$ can be considered blocked, and do not have to be expanded.

To improve savings during search, we would of course like to increase the number of pruned nodes, i.e., increase the number of nodes inside the swamp. On the other hand, a swamp that itself contains the start or target state cannot be pruned, and thus larger swamps will be used less often. For example, as the definition of swamps implies, the entire graph is a trivial swamp, but is useless when pruning states during *any* search. Thus, there is a tradeoff between a swamp's size and its usability. To handle this issue, we detect a set of small contiguous-swamps that cover as much of the graph as possible, but can be also used together. We call such sets *modular-swamps*. Later in the paper we extend the notion of modular-swamps to the more complex (but very useful) notion of *swamp-hierarchies*.

Modular-Swamps

It is important to note that two regions can be individual contiguous-swamps without their union being a swamp, as shown in the following example.

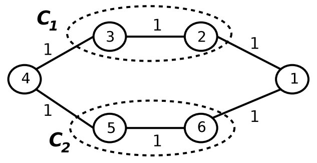


Figure 2: Two contiguous-swamps; Their union is not a swamp.

Example 2. *Figure 2 depicts a graph with two contiguous-swamps, $\mathcal{C}_1$ and $\mathcal{C}_2$. The reader can verify that, while the individual contiguous-swamps appropriately satisfy the definition of swamps, their union does not—any path between states 1 and 4 must pass through one of the two swamps.*

The example above motivates us to define a modular-swamp as a set of contiguous-swamps that can be used together effectively.

Definition 3. *A modular-swamp $\mathcal{M}$ is a set of disjoint contiguous-swamps $\mathcal{M} = \{\mathcal{C}_1, \dots, \mathcal{C}_k\} \quad \forall i \neq j \quad \mathcal{C}_i \cap \mathcal{C}_j = \emptyset$, such that any subset of them forms a swamp. That is, if $\mathcal{M}' \subseteq \mathcal{M}$ then $\bigcup_{\mathcal{C} \in \mathcal{M}'} \mathcal{C}$ is a swamp.*

Within the context of modular-swamp $\mathcal{M}$, let $\mathcal{C}_{\mathcal{M}}(v)$ denote the contiguous-swamp within $\mathcal{M}$ that contains state v.

$$\mathcal{C}_{\mathcal{M}}(v) \triangleq \begin{cases} \mathcal{C}_i \in \mathcal{M} & \text{for which } v \in \mathcal{C}_i \\ \emptyset & \text{if } v \text{ is not contained in any} \\ & \text{contiguous-swamp within } \mathcal{M}. \end{cases}$$

We will usually discard the subscript $\mathcal{M}$ when the set $\mathcal{M}$ with which we are operating is obvious from the context.

Definition 3 provides a natural way to conduct a search over a graph, within which a modular-swamp $\mathcal{M}$ is known: whenever we search for a path between states v_1, v_2, we will block access to all contiguous-swamps in $\mathcal{M}$, except perhaps $\mathcal{C}(v_1)$ and $\mathcal{C}(v_2)$—the contiguous-swamps that contain nodes v_1 and v_2. The remaining contiguous-swamps form a swamp together, and can thus be considered blocked for the purpose of the search (which, through our construction, originates and ends outside the swamp). The real challenge, though, is to *find* modular-swamps in the graph (we present an algorithm below).

Since modular-swamps are restricted so as to contain only small contiguous-swamps that can be combined consistently, we will usually have difficulty finding a modular-swamp that covers the entire graph. We address this with a hierarchical approach that allows us to greatly expand the portion of the graph that is covered by contiguous-swamps.

Swamp-Hierarchies

Once we find a modular-swamp, we can recursively search for another modular-swamp on the remaining graph. This stage introduces dependencies between contiguous-swamps. Before formally defining these dependencies, we will illustrate them, with an example.

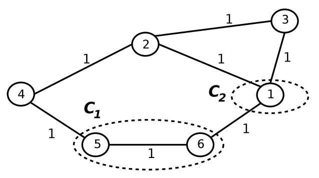


Figure 3: An example of a swamp-hierarchy

Example 3. *Figure 3 depicts a graph with two contiguous sets of states, $\mathcal{C}_1$ and $\mathcal{C}_2$. $\mathcal{C}_1$ is a contiguous-swamp that is composed of states 5 and 6. The reader can easily verify that these two states do not participate in the shortest path between nodes on the external boundary of $\mathcal{C}_1$.*

On the other hand, $\mathcal{C}_2$ is not a contiguous-swamp if considered on its own (it is, in fact, on the shortest path between state 6 and state 3). However, if the nodes in $\mathcal{C}_1$ are considered blocked, then $\mathcal{C}_2$ is a contiguous-swamp; that is, if we prune states 5 and 6 from the graph, then we can consider $\mathcal{C}_2$ a swamp.

We say that $\mathcal{C}_2$ above depends *on $\mathcal{C}_1$. Whenever we search for a shortest path on the graph, we can safely consider $\mathcal{C}_1$ as blocked (unless our search starts or ends inside it), and (given that $\mathcal{C}_1$ is considered blocked) we will consider $\mathcal{C}_2$ as blocked unless our search begins or ends in $\mathcal{C}_1 \cup \mathcal{C}_2$.*

To generalize the example above, we will define a partial order $\preceq$ on contiguous-swamps: intuitively, $\mathcal{C}_1 \preceq \mathcal{C}_2$ if $\mathcal{C}_2$ depends on $\mathcal{C}_1$, in the same manner as the example above.

We remind the reader that a partial order $\preceq$ is

- reflexive ($\mathcal{C}_i \preceq \mathcal{C}_i$),
- antisymmetric (if $\mathcal{C}_i \preceq \mathcal{C}_j$ and $\mathcal{C}_j \preceq \mathcal{C}_i$ then $\mathcal{C}_i = \mathcal{C}_j$)
- and transitive (if $\mathcal{C}_i \preceq \mathcal{C}_j$ and $\mathcal{C}_j \preceq \mathcal{C}_l$ then $\mathcal{C}_i \preceq \mathcal{C}_l$).

We define the closure of a set of contiguous-swamps $T \subseteq \mathcal{H}$ under $\preceq$ as the following set:

$$T^{\preceq} = \{\mathcal{C} \in \mathcal{H} \mid \mathcal{C} \preceq \mathcal{C}' \text{ for some } \mathcal{C}' \in T\}$$

That is, $T^{\preceq}$ is the set T extended with all contiguous-swamps on which members of T depend.

We are now ready to formally define a swamp-hierarchy:

Definition 4. *A swamp-hierarchy in a graph G is a tuple $(\mathcal{H}, \preceq)$ where $\mathcal{H}$ is a set of disjoint contiguous-swamps:*

$$\mathcal{H} = \{\mathcal{C}_1, \dots, \mathcal{C}_k\} \quad \forall i \neq j \quad \mathcal{C}_i \cap \mathcal{C}_j = \emptyset$$

and $\preceq$ is a partial order on them such that the closure of any subset of contiguous-swamps from $\mathcal{H}$ forms a swamp in G; i.e., $\forall\, T \subseteq \mathcal{H}$ we have that

$$\mathcal{S} = \bigcup_{\mathcal{C} \in T^{\preceq}} \mathcal{C} \quad \textit{is a swamp in } G$$

The definition of a swamp-hierarchy may seem somewhat complex, but using it to effectively prune the graph is quite simple. The following observation highlights the usefulness of the definition.

Observation 1. *Given that we are searching for a path between states v_1 and v_2 in a graph for which we have a swamp-hierarchy $(\mathcal{H}, \preceq)$, we can remove from the set $\mathcal{H}$ the contiguous-swamps that contain v_1, v_2, and any contiguous-swamp that depends on them. This set is closed under $\preceq$.*

$$T = \mathcal{H} \setminus \{\mathcal{C} \,:\, \mathcal{C}(v_1) \preceq \mathcal{C} \,\vee\, \mathcal{C}(v_2) \preceq \mathcal{C}\}$$

$$\textit{we therefore have } T^{\preceq} = T$$

Therefore, by Definition 4, the union of all contiguous-swamps in T is a swamp, and we can consider it blocked when searching between v_1 and v_2 (specifically, v_1 and v_2 are not contained in any of the contiguous-swamps in T).

A swamp-hierarchy will thus be most effective if it covers as much of the search space as possible, and contains as few dependencies as possible. This way, a large portion of the graph will be blocked in all searches.

Constructing Swamps

Finding a Single Contiguous-Swamp

We begin our search for a contiguous-swamp with a state around which we would like to find a swamp. We call this state a *seed*. Later, we will search for swamps around a set of seeds in different locations in the graph, in order to try and cover it with as many contiguous-swamps as possible.[2]

According to Lemma 1, all we need to do in order to confirm that a set of states is a contiguous-swamp is to check that there exists a shortest path between any pair of states on the boundary of the set that does not pass through the suspected set. Note that checking this is usually fast if the set of nodes is small, as the searches are very local.

To generate a candidate set for the detection process, we take the set of states with distance less than or equal to radius r from the seed. If a problem is found during the confirmation of the set (in the form of a pair of points (x, y) on the boundary of the set for which no alternative shortest path exists), the following *trimming process* is performed.

We keep the set of states that are connected to the seed after removing the states that are on the shortest path between x and y. We then repeat the confirmation process with the *trimmed set* (of the remaining states). We continue to try other candidate sets (of larger and larger radii around the seed) and keep the largest contiguous-swamp that was found around the seed.

The trimming process is described formally[3] in Alg. 1.

Algorithm 1 Trim to Swamp

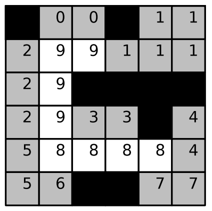
```
procedure TRIM($stateSet, Graph$)
    repeat
        $trimmed := false$
        $B := getBoundary(stateSet, Graph)$
        for all $v_1, v_2 \in B$ do
            $P_1$ := findPath($v_1$, $v_2$, $Graph$)
            $P_2$ := findPath($v_1$, $v_2$, $Graph \setminus stateSet$)
            if $length(P_2) > length(P_1)$
                $stateSet := stateSet \setminus P_2$
                $keepConnected(stateSet, seed)$
                $trimmed := true$
    until $\neg trimmed$
return $stateSet$
```

Constructing Modular-Swamps

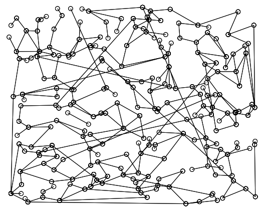
Given the procedure for detecting a single contiguous-swamp, we would like to find a set of contiguous-swamps that we can use together. As shown in Example 2, a union of arbitrary contiguous-swamps is not guaranteed to form a swamp. Instead, our algorithm will start with an empty modular-swamp, and will iteratively add contiguous-swamps in a way that preserves the modularity of the set.

[2]On a general graph, all states are potential seeds. Sometimes, with domain-specific knowledge, we can consider a smaller set of states; e.g., it is often possible to reduce the set of seeds to states adjacent to obstacles, and corners are natural seeds as well. We will not address the issue in this paper due to space constraints.

[3]For simplicity of presentation, and because the implementation is straightforward, we use the functions getBoundary() which returns the boundary of a set of states, and keepConnected() which keeps only the subset of states that is connected to the seed, without showing their implementation.



(a)



(b)

Figure 4: Examples. (a) Grid having a swamp hierarchy with 2 levels. Contiguous-swamps are numbered and colored—gray for level zero, and white for level one. (b) Delaunay Graph with 250 vertices.

The algorithm we use checks that the candidate set of states S will form a swamp when added to any subset of the current modular-swamp $\mathcal{M}$: we run two searches between any pair of states on the boundary of S. The first search is executed while considering $\mathcal{M}$ as a modular-swamp (see details below), and the second search is done considering $\mathcal{M} \cup S$ as a modular-swamp. If each pair of searches gives a solution with the same cost, we can define the new modular-swamp as $\mathcal{M} \cup S$. This (along with the same trimming method we have shown for regular contiguous-swamps) is depicted in Algorithm 2.

Algorithm 2 Trim to Consistent Swamp

```
procedure FINDCONSSWAMP($stateSet, G, \mathcal{M}$)
    repeat
        $trimmed := false$
        $B := getBoundary(stateSet, G)$
        for all $v_1, v_2 \in B$ do
            $OpenG := G \setminus (C \setminus \{\mathcal{C}_{\mathcal{M}}(v_1), \mathcal{C}_{\mathcal{M}}(v_2)\})$
            $P_1$ := findPath($v_1$, $v_2$, $OpenG$)
            $P_2$ := findPath($v_1$, $v_2$, $OpenG \setminus stateSet$)
            if $length(P_2) > length(P1)$
                $stateSet := stateSet \setminus P_2$
                $keepConnected(stateSet, seed)$
                $trimmed := true$
    until $\neg trimmed$
return $stateSet$
```

Note that Alg. 2 did not exhaustively check that every subset of the new modular-swamp is a swamp. Still, the following theorem states that this property does, indeed, hold.

Theorem 2. *Given a modular-swamp $\mathcal{M}$, the output of Algorithm 2 gives a set of states $\mathcal{C}$ such that $\mathcal{M} \cup \{\mathcal{C}\}$ is also a modular-swamp.*

Figure 4(a) illustrates the results achieved by our algorithm for constructing modular-swamps. The numbers on the gray squares are the different contiguous-swamps. The union of all these contiguous-swamps is a modular swamp. The numbers in the white cells represent another level in a swamp hierarchy which we shall now show how to detect.

Constructing Swamp-Hierarchies

We now present a simple way to construct swamp-hierarchies. While this is only one of many possible methods for doing it, this method has the advantage of relying on local dependencies between contiguous-swamps, which keeps the method simple and fast.

We start by finding a modular-swamp using the technique described above. The contiguous-swamps of this modular-swamp are considered as level zero of the hierarchy. Then, we construct another modular-swamp, but with a slight change: this time, all states that are in contiguous-swamps that were detected at lower levels are considered to be blocked. We continue this iterative process until we are no longer able to produce a non-empty modular-swamp.

We denote the modular-swamp we found during the i'th iteration by $\mathcal{M}_i$. We then define the following swamp-hierarchy: $\mathcal{H} = \bigcup_i \mathcal{M}_i$. We define the dependency structure so that contiguous-swamps from a higher level in the hierarchy depend on a contiguous-swamp from a lower level if they are touching, i.e., if one has nodes that are on the boundary of the other. Formally:

- $\mathcal{C}_j \preceq \mathcal{C}_i$ if $\mathcal{C}_i \in \mathcal{M}_k$ and $\mathcal{C}_j \in \mathcal{M}_l$ for some $k > l$ and there exists a state $s \in B(\mathcal{C}_i)$ such that $s \in \mathcal{C}_j$ or,
- if transitivity implies the above. That is, $\mathcal{C}_i \preceq \mathcal{C}_k$ if there exists $\mathcal{C}_j$ such that $\mathcal{C}_i \preceq \mathcal{C}_j$ and $\mathcal{C}_j \preceq \mathcal{C}_k$.

Theorem 3. *$\mathcal{H} = \bigcup_i \mathcal{M}_i$, along with the partial order $\preceq$ as defined above, defines a swamp-hierarchy.*

Memory Consumption The representation of our swamp hierarchy in memory is quite efficient. For each state that is part of a swamp, we need to specify its contiguous-swamp. In addition, each contiguous-swamp keeps a list of contiguous-swamps on which it depends (among its neighbors). This amount of memory can accumulate to slightly more than $|V|$. Other methods such as TDH achieve their full benefit only when much more memory is available.

Experimental Results

To explore our method's effectiveness, we ran experiments in several domains. For each graph, we ran our swamp detection algorithms, and conducted A^* searches between 10,000 randomly selected pairs of states. For comparison, searches were also conducted without swamps.

As time measurements are implementation and machine-specific, we also examined the number of nodes expanded during search. It is still important to examine time measurements, so as to make sure that the overhead caused by using swamps during run-time (i.e., blocking off certain areas) does not negate the advantage gained by lowering the number of expanded nodes.

The domains we used were random grids, random graphs, Delaunay graphs, random mazes, room maps, and maps from the computer game Baldur's Gate.

For grid-based domains we assumed 8-neighbor connectivity (a diagonal move's cost is $\sqrt{2}$), and used the octile distance heuristic; for Delaunay Graphs, we used the Euclidean distance heuristic. For random graphs no heuristic exists. Results on each domain are described below, and summarized in Table 1; in all these domains with the exception of random graphs (which are particularly difficult), swamps enabled considerable reduction in both the number of nodes expanded by A^*, as well as search run-time.

Mazes We generated squared mazes of different sizes using a random version of Prim's algorithm. Figures 5(a), 5(b), and 5(c) demonstrate the effectiveness of swamps in the maze domain. Figure 5(a) shows the number of nodes A^* expanded when using and not using swamps, compared to the number of nodes on the shortest path (the lower bound of nodes that must be expanded). Using swamps caused A^* to expand only a few more nodes than the lower bound; without swamps, A^* expanded a much larger number. The benefit from swamps increases with the size of the maze. Figure 5(b) shows a similar trend for average search time with and without swamps (see also Table 1).

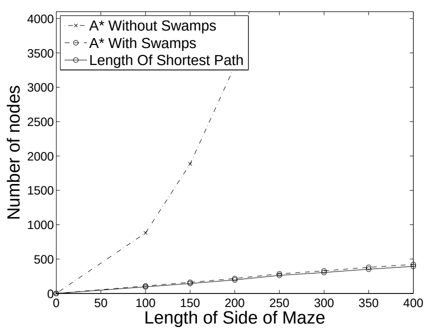
While preprocessing time is usually significant with other search improvement techniques that rely on preprocessing (e.g., those in the related work section), detection of swamps can be relatively quick. Figure 5(c) shows the number of searches it takes on average to recover the cost of the swamp detection process. The results showed that on large mazes of size 400×400, the preprocessing cost is recovered in full after fewer than 400 searches.

Room maps Room maps consisted of 256×256 grids, with rooms of size 32×32, and random doors between them (maps came from HOG (Sturtevant 2009)). Results are shown in Table 1. While the octile heuristic is more accurate in rooms than in mazes, using swamps still gave a significant advantage in both the number of expanded nodes and the time of the search, as shown in the table.

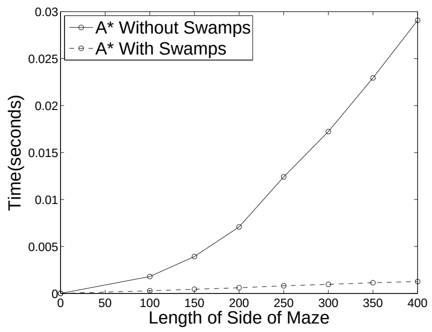
Baldur's Gate maps This domain consists of maps from the computer game Baldur's Gate. These maps showed high variability of results, because of the different structure of each map; our experiments included all maps of size greater than 100×100. Results (in Table 1) show that using swamps saved a factor of almost 3 in node expansion. The results for search time are almost as good, as the check for swamps requires only small overhead. Note that most maps in Baldur's Gate are relatively small compared to maps that are currently being used by more modern computer games, and we expect that with larger maps, greater savings will be achieved.

Random Grids This domain consists of 8-connected grids, with randomly placed obstacles. This would seem to be the worst-case scenario for swamps on a grid where obstacles exist; the reason is that structures are easily utilized by swamps, while random obstacles mean that useful structures may not appear. Our experiments show that even on random grids, the hierarchal swamp approach can improve search efficiency. We also witnessed the phenomenon that the improvement is greater as the graph gets larger. Results for grids of size 300×300 are shown in Table 1. Even in this domain, using swamps significantly reduced both the number of expanded nodes, and the average search time.

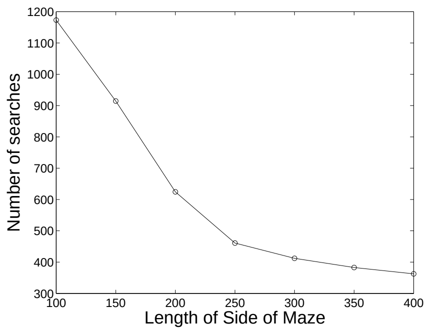
Delaunay Graphs Delaunay graphs have been used in the past to simulate transportation networks (Felner, Stern, and Kraus 2002); a randomly generated set of points on a 2D plane is made into a graph using Delaunay triangulation. As a result, nodes are locally connected. To simulate a real



(a)



(b)



(c)

Figure 5: (a) Nodes expanded with and without using swamps, and length of shortest path; (b) Search time with and without swamps; (c) Number of searches it takes until the time gained in searches from using swamps equals preprocessing costs.

	Nodes	Time	Swamp%	#Searches
Mazes (400×400)	3.41%	4.36%	100%	362
Rooms	15.27%	13.53%	88.40%	865
Baldur's Gate	34.58%	41.18%	73.58%	874
Random Grids	32.04%	33.45%	79.26%	2111
Delaunay Graphs	47.83%	42.70%	57.84%	1011
Random Graphs	94.12%	94.61%	6.27%	2319600

Table 1: Results comparing A^* performance with and without swamps: Nodes—the percentage of nodes expanded using swamps, compared to A^* without swamps; Time—time it took to search with swamps as a percentage of the time it took to search without swamps; Swamp%—the percentage of states that are part of some contiguous-swamp; #Searches—number of searches it took to recover detection cost.

roadmap, in which not all close points are connected, some edges are also randomly removed. Random edges are added to represent highways. For the results presented here, we removed 60% of the edges, and added edges such that every vertex has a 5% chance to be connected to a highway (see Figure 4(b) for an example). The heuristic used for search was Euclidean distance. Even in this domain, almost 58% of the graph was covered with swamps, which led to substantial savings: A^* with swamps only expanded 48% of the nodes that it would have expanded without swamps, and search time was 43% of the search time of regular A^*.

Random Graphs Random graphs ($G(n,p)$) are graphs with n vertices, where an edge between any pair of vertices exists with probability p. We generated random graphs with $n = 800$ and probability $p = \frac{log(n)}{n} + \epsilon$, which is the threshold for an almost-surely connected graph. Edges had random weight between 1 and the number of vertices. As the graph is random and not planar, no heuristic exists for it—so we used A^* with a heuristic value of 1 for non-goal states, and 0 for a goal state. As the graph has a high degree using p, we did not expect it to have many useful swamps. However, we did manage to detect 6.27% of the graph as swamps, saving 5.88% on node expansion and 5.39% on search time.

Conclusions

We presented the concept of *swamps*, sets of nodes in a graph that can needlessly slow search. We defined modular-swamps, and swamp-hierarchies, presenting algorithms that make use of their properties to reduce search costs while still detecting optimal paths. We presented an algorithm that detects swamps, and applied it to various domains. Our experiments show that the gains from utilizing swamps can be substantial, reducing both the number of nodes expanded during search, and the time it takes to perform searches.

Acknowledgments

This research was supported in part by Israel Science Foundation grants #898/05 and #305/09.

References

Björnsson, Y., and Halldórsson, K. 2006. Improved heuristics for optimal path-finding on game maps. *AIIDE* 9–14.

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

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Demyen, D., and Buro, M. 2006. Efficient triangulation-based pathfinding. In *AAAI'06*.

Felner, A.; Berrer, M.; Sturtevant, N.; and Schaeffer, J. 2009. Abstraction-based heuristics with true distance computations. In *Proceedings of SARA-09*.

Felner, A.; Stern, R.; and Kraus, S. 2002. PHA*: performing A^* in unknown physical environments. In *AAMAS '02*, 240–247. New York, NY, USA: ACM.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental Algorithms (WEA 2008)*, 319–333.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2009. Using swamps to improve optimal pathfinding (extended abstract). In *AAMAS 2009*, 1163–1164.

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

Sturtevant, N.; Felner, A.; Barer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *IJCAI-09*, 609–614.

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

Sturtevant, N. R. 2009. HOG, hierarchical open graph. http://code.google.com/p/hog2/source/checkout.