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.[1] 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 a 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.

[1] A full version of this work appears in (Pochter et al. 2010).

Swamps

A swamp is formally defined as follows:

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 $\mathcal{C}$.

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.

Modular Swamps

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. To handle this issue, we detect a set of small contiguous-swamps that cover as much of the graph as possible, but can also be used together. We call such sets *modular-swamps*.

Definition 2. *A modular-swamp* $\mathcal{M}$ *is a set of disjoint contiguous-swamps* $\mathcal{M} = \{\mathcal{C}_1, \ldots, \mathcal{C}_k\} \quad \forall i \neq j \quad \mathcal{C}_i \cap \mathcal{C}_j = \emptyset$*, 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 $\mathcal{C}(v)$ denote the contiguous-swamp within $\mathcal{M}$ that contains state v.

Definition 2 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.

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, as illustrated by the following example:

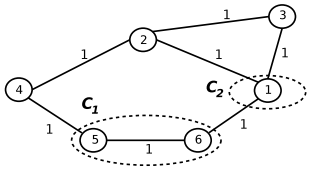

Figure 1: An example of a swamp-hierarchy

Example 1. *Figure 1 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 outside 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.

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

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

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

For a more detailed description of the algorithms used for detecting swamps, see (Pochter et al. 2010).

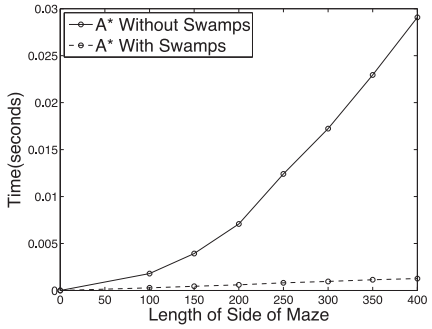

Figure 2: Search time on mazes with and without swamps

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

Experimental Results

To explore our method's effectiveness, we ran experiments in several domains: random grids, random graphs, Delaunay graphs, random mazes, room maps, and maps from the computer game Baldur's Gate. 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.

For each domain we examined the average number of expanded nodes, the average time per search and the number of searches it took to recover the computational costs of the preprocessing stage. 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 are 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. For some domains, savings increase dramatically as the graph grows (see Figure 2).

Acknowledgments

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

References

Pochter, N.; Zohar, A.; Rosenschein, J. S.; and Felner, A. 2010. Search space reduction using swamp hierarchies. In *The Twenty-Fourth National Conference on Artificial Intelligence*. To appear.