Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

State Space Searching

State space searches are used in everything from network routing to games. David implements a C++ library for performing state space searches.

October 01, 2004
URL:http://www.drdobbs.com/state-space-searching/184401857

David Theese works at Siebel Systems. He can be contacted at dtheese@yahoo.com.

Suppressing Repeated State Generation

The need for state space search occurs in many problem domains, including network routing (and route finding in general), parsing, VLSI design, and game playing. This wide range of usefulness motivated me to develop a generic library for performing state space searches. In this article, I provide an overview of state space search, describe how to use my library, and touch on a couple of implementation details.

In designing the library, I used a policy-based approach [1]. This allows easy configuration of the library by the client and, since policy-based design is a template-based technique, the benefits of type safety are realized while still achieving the desired type (problem domain) independence. Although the library is implemented purely in Standard C++, some compilers might have a problem with it due to heavy template usage. Complete source code for the library is at http://www.cuj.com/code/. In addition, I've included a sample client that solves the toy sliding tile problem [2]. The library is implemented in the self-contained files search.h and search.inl, respectively.

State Space Search Overview

State space search is used for problem solving. A state space is the set of all discrete configurations (that is, states) that might be encountered while trying to solve a problem. A state space search entails systematically exploring these states in an effort to find a solution to the problem. Sometimes, any solution will do. But frequently, a solution that can be reached with least possible cost (an optimal solution) is desired.

Consider, for example, the sliding tile problem. A solution to this problem is an ordered list of moves that, when applied to the initial state, results in the tiles being in the desired order. Depending on the search algorithm used, the solution may or may not be optimal.

To search a state space, you need to be able to move between states. This is done via the application of operators. (In the context of the sliding tile problem, an operator is a command directing the movement of a specific tile.) This takes you from some state to some other adjacent state. The search proceeds by systematically generating all operators that may be applied to the initial state, applying those operators to expand the initial state into its child states, repeating the process on those states, and so on. This continues until you either succeed (that is, find a goal state) or exhaustively check the entire state space and find that no goal state exists (be wary of infinite state spaces that may not contain a goal state!).

While searching a state space, you will likely encounter the problem of states being repeatedly generated. This can introduce extreme inefficiency to the search, as it causes you to explore fruitless paths multiple times. Several possibilities for dealing with this problem (from least to most effective) are suggested on page 82 of [3]:

1. Don't return to a state's parent state.
2. Don't return to any of a state's ancestors.
3. Don't return to any state ever previously generated.

There are many different algorithms for searching a state space, and the primary difference between those presented here lies in the order in which states are expanded. One way to categorize these algorithms is by whether a heuristic [4] is used in determining which state to expand next. Since a heuristic uses problem-specific knowledge, algorithms that employ a heuristic are referred to as "informed." Those that don't are referred to as "uninformed." For more details, see Chapters 3 and 4 of [3].

Uninformed Search

The library I present implements five uninformed search algorithms:

- **Breadth-first**, in which all of a state's children are expanded before any of their children are expanded.
- **Depth-first**, in which children are recursively expanded before their siblings.
- **Depth-limited**, depth-first that is prohibited from considering states beyond a specified depth limit.
- **Iterative deepening**, a repeated depth limited performed with a depth limit of 1, then with a depth limit of 2, and so on, up to a specified depth limit.

The iteration continues until a goal state is found or the depth limit is reached.
- **Uniform cost**, in which states are expanded in increasing order of the cost incurred to reach them.

These algorithms offer different trade-offs with regard to time/space requirements, whether they are guaranteed to find an optimal solution, or whether they are even guaranteed to find a solution at all (if one exists). In short, the following hold if a solution exists: Depth-first is not guaranteed to find a solution at all (allowing no repeated state generation will fix this if the state space is finite; but even then, the solution may not be optimal); depth-limited is guaranteed to find a solution as long as the depth limit is at least that of the shallowest solution, but it may not be optimal (see the sidebar entitled "Suppressing Repeated State Generation"); in all other cases, an optimal solution is guaranteed to be found.

Informed Search

The library implements two informed search algorithms:

- **Greedy** search, in which states are expanded in increasing order of their estimated cost (as judged by the heuristic function) to reach the nearest goal state.
- **A***, which is the most effective search algorithm of those presented here. It is a combination of **uniform cost** and **greedy** search. For every state **S** being considered for expansion, it adds the estimated cost (as judged by the heuristic function) from **S** to the goal to the actual cost to get from the initial state to **S**. This is an estimate of the cost to get from the initial state to the goal state via **S**. States are expanded in increasing order of this estimated cost.

In short, the following hold if a solution exists: Though **greedy** search does have a tendency to find solutions quickly, it is not guaranteed to find a solution at all (allowing no repeated state generation will fix this if the state space is finite; but even then, the solution may not be optimal). In contrast, **A*** is guaranteed to find an optimal solution so long as one condition is met. The heuristic employed must be admissible—it must never overestimate the cost to reach the nearest goal state.

Using the Library

To use the library, clients must instantiate **search_engine_t**, whose declaration is:

```
template
<
  class STATE_T, class OP_T,
  class OP_GEN_T, class OP_APPLY_T,
  class GOAL_TEST_T,
  class REPEATED_STATE_CHECKER_T,
  class HEURISTIC_T = null_heuristic_t
> class search_engine_t;
```

STATE_T is a user-defined type representing a state. **OP_T** is a user-defined type representing an operator. They must:

- Be default constructible.
- Be copy constructible.
- Be assignable.
- Be destructible.
- Implement **operator<()**.

OP_GEN_T, **OP_APPLY_T**, and **GOAL_TEST_T** are user-defined policy classes that must provide nonprivate member functions with the following semantics and signatures:

```
OP_GEN_T:
// Return all operators that may
// legally be applied to a state
std::set<OP_T>
gen_ops(const STATE_T &);

OP_APPLY_T:
// Return the resulting state and
// cost of application of an
// operator to a state
std::pair<STATE_T, double>
apply_op(const STATE_T &,
         const OP_T &);
GOAL_TEST_T:
// Check whether a given state is a
// goal state
bool
is_goal(const STATE_T &);
```

REPEATED_STATE_CHECKER_T should be substituted with one of the five library-supplied full specializations of class template **repeated_state_checker_t<>**. The five choices are:

- **repeated_state_checker_t<0>**. No repeated state checking (generally ill-advised).
- **repeated_state_checker_t<1>**. Option 1 described earlier.
- **repeated_state_checker_t<2>**. Option 2 described earlier.

- **repeated_state_checker_t<3>**. Option 3 described earlier.
- **repeated_state_checker_t<4, CHECK_T>**. Use a user-defined repeated state checking policy **CHECK_T**.

If specialization #4 is used, **CHECK_T** must be supplied (it may be omitted otherwise due to the presence of a default template argument). It must provide nonprivate member functions with the following semantics and signatures:

```
// Clear all internally stored
// states, if any
void clear_states();

// Check if a given state has been
// seen before
bool seen_before(const STATE_T &);
```

Clients would commonly want to use specialization #4 if they have developed a hashing scheme for their **STATE_T** type (allowing constant lookup time). Specialization #3 provides full repeated state checking, but it does so using an **std:set<>**, which provides logarithmic lookup time.

HEURISTIC_T need only be supplied if one of the informed search algorithms (**Greedy** search or **A***) is used (it may be omitted otherwise due to the presence of a default template argument). If supplied, it must provide a nonprivate member function with the following semantics and signature:

```
// For a given state, return the
// estimated cost from that state
// to the nearest goal state.
double
apply_heuristic(const STATE_T &);
```

Once **search_engine_t<>** has been instantiated, the member function corresponding to the desired search algorithm should be invoked. Each member function takes an initial state from which to start the search. Where applicable, some member functions take a depth limit and, in all cases except **iterative_deepening()**, it is possible to pass a flag indicating that all solutions, rather than just the first found, are to be returned [5]. Each of these member functions returns an **std::set<>** of solution structures. Each structure contains a goal state reached, the cost to get to it, and the operator sequence leading to it. See Listing 1 for the relevant declarations. Listing 2 provides a brief demonstration of use of the library.

The library neither generates nor traps exceptions. Any exceptions that propagate out of the library will have originated from either the Standard Library or user-supplied code.

Storage of the Search Tree

During library implementation, I found myself needing to create an arbitrary **n**-ary tree to store the search tree that is built up as states are expanded, and I sought to find a way to store it that did not involve explicit memory management (and the associated maintenance problems it tends to spawn).

My solution was to store the vertices of the tree as elements in an **std::list<>**. A vertex's parent/child pointers simply point to other elements in the list. The list is never used as such—it is not traversed, sorted, spliced, and so on. It is just a place to store vertices and a mechanism to ensure they get deallocated automatically. When the list is destroyed, the entire tree is also automatically destroyed in one fell swoop.

I chose **std::list<>** for a specific reason: The insertion of new elements will not invalidate pointers to other elements. A vertex can always be assured its parent/child pointers will remain valid. In addition, the insertion of new elements occurs in constant time.

One Search Function Underlying all Algorithms

Again, the primary difference between the search algorithms is the order in which states are expanded. For depth-first, depth-limited, and iterative deepening, new states can be pushed onto an **std::stack<>**. For breadth-first, new states can be inserted into an **std::queue<>**. For the remaining algorithms, new states can be inserted into an **std::priority_queue<>** created with an appropriate ordering criterion.

It is obviously desirable to have one search function underlying each of the algorithms (and this was accomplished). Providing, as a function template argument, one of the three container types aforementioned as the data structure to store the states-to-be-expanded in almost allows this. All three support **push()**, **pop()**, and **empty()**, but only **std::stack<>** and **std::priority_queue<>** support **top()**. The equivalent in **std::queue<>** has a different name—**front()**. Since these three class templates don't have a consistent interface, I couldn't quite get away with just using the needed member functions directly. Oh, so close! I was forced to deal with the **top()** versus **front()** issue.

It is tempting to use **push()**, **pop()**, and **empty()** directly and to supply a pointer-to-member to either **top()** or **front()**. However, this approach is problematic. To declare a pointer-to-member, an exact member function signature must be known. Believe it or not, the signatures for **top()** and **front()** are not precisely specified in the C++ Standard! We know these member functions return a reference to an element and that they should be called with no arguments. But we don't know that they take no arguments—it is possible they have arguments, all of which have a default value. This provision is specifically spelled out in the C++ Standard [6].

My solution was to supply the container type as a function template argument, use **push()**, **pop()**, and **empty()** directly, and then deal with **top()** versus **front()** as follows: I created a **get_next_node_to_expand()** member function, taking a reference to **std::queue<>**. This member function is just a pass-through to **front()**. I also created a templated **get_next_node_to_expand()** member function, taking a reference to any container type. This member function is just a pass-through to **top()**. A simple call to **get_next_node_to_ expand()** is made, passing in a reference to the container (whatever its type), and overload resolution solves the problem for us. Voilà!

Acknowledgments

Thanks to Ryan Stephens, Bret Carruthers, and John Ericson for providing peer reviews of the article text and source code.

References

[1] Alexandrescu, Andrei. *Modern C++ Design*, Addison-Wesley, 2001.

[2] In this well-known problem, movable square tiles are placed within a constraining square frame. One space is left empty, and tiles adjacent to this empty space may slide into it. By sliding tiles in this manner, you try to get the tiles into some particular arrangement. An interesting fact about this problem is that its state space is split evenly into two disjoint sets. A randomly chosen instance of this problem has exactly a 50 percent chance of being solvable!

[3] Russell, Stuart and Peter Norvig. *Artificial Intelligence: A Modern Approach*, Prentice-Hall, 1995.

[4] In this context, a heuristic is a function that, given a state, will make an estimate of the cost to get from that state to the nearest goal state.

[5] If the client desires to find paths to all goal states (of which there may be any number), this is done by recording the existence, and path to, any goal state found, then continuing the search in a normal manner. This continues until the entire state space has been searched (this makes the choice of search algorithm irrelevant when searching for all solutions). For obvious reasons, this should not be attempted with problems that have an infinite state space! You should also be sure to completely suppress the generation of repeated states when exhaustively searching a state space.

When using iterative deepening, it does not make sense to search for all goal states, and so this is not supported. By definition, this algorithm searches only until the first goal state is encountered.

In this article, the assumption is made that the client desires only a path to one goal state. For most problems, the state space will simply be too large to search exhaustively.

[6] Sutter, Herb. Guru of the Week #64 (http://www.gotw.ca/gotw/064.htm).

Listing 1

```
// search.h
namespace dct
{
   template <typename STATE_T, typename OP_T>
   struct search_result_template_t
   {
      STATE_T           goal_state_reached;
      double            cost;
      std::vector<OP_T> operators;
   };
   template <...> // See article text for template parameters
   class search_engine_t: ... // Inherit from policy classes
   {
      public:
         typedef std::set<
                         search_result_template_t<STATE_T, OP_T>,
                         result_comp_t<STATE_T, OP_T> // Not shown
                         > search_result_t;
         search_result_t a_star(
                                const STATE_T &initial_state,
                                bool          return_all = false
                               );
         search_result_t breadth_first(
                                const STATE_T &initial_state,
                                bool          return_all = false
                                     );
         search_result_t depth_first(
                                const STATE_T &initial_state,
                                bool          return_all = false
                                    );
         search_result_t depth_limited(
                                const STATE_T &initial_state,
                                unsigned int  depth_limit,
                                bool          return_all = false
                                     );
         search_result_t greedy(
                                const STATE_T &initial_state,
                                bool          return_all = false
                               );
         search_result_t iterative_deepening(
                                const STATE_T &initial_state,
                                unsigned int  depth_limit
                                           );

         search_result_t uniform_cost(
                                const STATE_T &initial_state,
                                bool          return_all = false
```

```
                                        );
   };
}
```

Listing 2

```
#include "search.h"

class state_t {...}; // A state
class op_t {...}; // An operator

class op_gen_t // Generate operators
{
   std::set<op_t> gen_ops(const state_t &state) {...}
};
class op_apply_t // Apply an operator to a state
{
   std::pair<state_t, double>
   apply_op(const state_t &state, const op_t &op) {...}
};
class goal_test_t // Check if a state is a goal state
{
   bool is_goal(const state_t &state) {...}
};
class check_t // User-defined repeated state checker
{
   void clear_state() {...}
   bool seen_before(const state_t &state) {...}
};
class heuristic_t // Estimate the cost to the nearest goal
{
   double apply_heuristic(const state_t &state) {...}
};
typedef dct::search_engine_t<state_t, op_t, op_gen_t,
                  op_apply_t, goal_test_t,
                  repeated_state_checker_t<4, check_t>,
                  heuristic_t> se_t;
int main()
{
   // Construct the initial state
   state_t initial_state(/* Constructor arguments */);
   se_t se;
   se_t::search_result_t solution;

   // Perform the search using A*, for example.
   solution = se.a_star(initial_state);

   // Now go do something with the solution...
}
```

Suppressing Repeated State Generation

There appears to be interesting interaction between depth-limited search (and, by extension, iterative deepening) and the suppression of repeated state generation. If option 3 is used for suppressing repeated states, it may be necessary to set the depth limit higher than the depth of the shallowest goal in order to reach that goal. Though I have not read of this in the literature, I have observed it empirically.

If the shortest path(s) to the goal **G** are at depth **N** and the depth limit is set to **N**, the following problem may occur. Suppose that on each path to **G**, some state **S** on that path at depth **M** is reached in greater than **M** steps purely because the optimal path to **S** was not the first path explored. You miss the goal because this suboptimal path through **S** is longer than the depth limit. But the real problem is that repeated state suppression prevents you from getting past **S** (and on to the goal) later when you do explore the optimal path to **S**. You are stopped at **S** because it has already been seen.

This problem brings to light the fundamental difference between options 2 and 3 for repeated state suppression. Option 2 only prevents cycles, whereas option 3 prevents both cycles and the exploration past **S** through two different paths (of potentially different length) leading to **S**.

—D.T.