

Djinni User's Guide

Robert J. Hansen <rjhansen@cs.uiowa.edu>

July 19, 2005

Contents

1	Legalese	
1.1	Copyright and Trademarks	
1.2	Authorship	
1.3	Intended Audience	
2	Concept Overview	
2.1	Boggle: Overview	
2.1.1	Game Explanation	
2.1.2	World	
2.1.3	Solution	
2.2	Engine	
3	Worlds	
4	Solution	
4.1	Required Methods	
5	Engine	
6	Runtime	
7	Putting It All Together	
8	Further Directions	
9	A Note On Names	

Abstract

1 Djinni is a framework for finding approxi-
1 mations of solutions to NP-complete prob-
2 lems. At present, Djinni only supports
2 the Traveling Salesman Problem with time
windows (TSPTW), but Djinni is built to
2 make it easy to extend to support other
2 problems such as the Vehicle Routing Prob-
3 lem with time windows (VRPTW)—or even
3 something as innocuous as playing a per-
3 fect game of Boggle. In this paper we
3 present an overview of the Djinni frame-
work, and extend Djinni to support the
3 popular game Boggle as an example of how
it may be extended.

1 Legalese

6 1.1 Copyright and Trademarks

6 Djinni is © 2004–5, the University of Iowa.
It's provided under the terms of the mod-
7 ified BSD license, which disclaims all war-
ranties. If it breaks, you get to keep both
7 pieces.

This document is (c) 2005, the University of Iowa. It's released under terms of the Creative Commons Attribution, Noncommercial, Share-Alike license¹.

The use of any trademarks (such as that of the word game Boggle) should not be construed as a challenge to those marks.

1.2 Authorship

This paper was written in 2005 by Rob Hansen <rjhansen@cs.uiowa.edu>, with input from Drs. Jeff Ohlmann and Barrett Thomas. I (slipping for a moment away from the neutral third person) was the principal designer of Djinni. I'm an old-school hacker, which will likely be evident in my writing style and vocabulary usage. Any language I think which may be unclear to some readers will be explained in footnotes.

1.3 Intended Audience

This paper was originally intended for University of Iowa students taking undergraduate or graduate courses in search methods. As such, this paper will assume some familiarity with formal mathematical writing.

Djinni is written in C++, with bindings available for a variety of scripting languages (Perl, Python and Tcl). This paper assumes a basic understanding of the C++ language, whatever C++ compiler is being used, how to write or edit a Makefile, and any one or more of the scripting languages.

¹<http://creativecommons.org/licenses/by-nc-sa/2.5>

Readers should understand object-oriented programming and what it means for a method to be pure virtual. Djinni makes extensive use of proxy methods, but understanding proxies is not strictly necessary to understanding Djinni.

2 Concept Overview

The central abstraction of Djinni is that of the database search. By representing all the necessary information to an NP-complete problem inside a matrix, we can represent solutions to the problem as paths through the matrix. This is essentially a database search, and is one of the fundamental ideas in artificial intelligence.

A *world*, which we will henceforth denote by W , is a particular instantiation of the parameters for a given NP-complete problem. This may be a two-dimensional matrix of travel times for TSPTW problems, a three-dimensional matrix of travel times for VRPTW problems, or a one-dimensional matrix of dictionary words in the case of Boggle.

A *solution*, which we will henceforth denote by S , is a particular path through W . All solutions are stored as a vector² of floating-point numbers.³

²In both the mathematical and programming senses: internally they're represented by C++'s `std::vector`.

³Floating-point numbers were selected due to the fact integers can be represented as floats, but floats cannot be represented as integers. While it's hard to imagine solutions with paths where

An *engine*, which we will henceforth denote by E , is an algorithm which takes an initial input S_{init} and from it derives or constructs S_{opt} , which is the optimal path through W . Please note that E has no inherent ability to see W ; if it needs to see W it has to go through S_x .

A *runtime*, which we will henceforth denote by R , may be thought of as the tuple $\langle W, S, E \rangle$. A runtime is responsible for parsing a parameters file, instantiating versions of W , S and E appropriate for those parameters, and serving as a controller to the operations of E .

2.1 Boggle: Overview

2.1.1 Game Explanation

Boggle is a game in which a square matrix M of size N is populated with random letters, one per cell, save for any cell containing Q, which is assumed to contain QU. The purpose of the game is to construct the most words possible containing three or more letters, starting from any point or points within M .

2.1.2 World

Our choice of W_b is simple: a two-dimensional matrix corresponding to M .

floating-point values are required, it is conceivable, and so our choice of floating-point parameters is insurance against future needs.

2.1.3 Solution

A solution S_b is a vector of words found within W_b . As can be inferred, our engine will construct S rather than derive S_b from an initial value.

2.2 Engine

Our engine E_b will use S_b to get a picture of W_b , and will then construct via exhaustive search a solution S_{best} containing all the words to be found within W_b .

3 Worlds

Looking over the API reference for Djinni, we see that our World class constructor takes a single argument, *identifier*. This is used only by derived classes, such as the one we're about to write, and allows us to keep a short identifying string⁴ which can be displayed in the eventual output.

However, World by itself has no way of inputting data, and by default it stores all information as floating-point values and not characters. This isn't a problem: we just note that Djinni is not a perfect fit for the Boggle problem, and go on.⁵ In order to use

⁴If you want your identifying string to be *War and Peace*, don't let us stop you: it'll let you do that. However, brevity is the soul of wit.

⁵Djinni is not meant to be a perfect fit for all problems; it is only meant to be a workable fit for many problems. Using Boggle as an example will hopefully show how Djinni can be made into a workable fit for problems for which it was never designed.

the World class productively, we will have to do the following:

1. Write a function allowing us to read in character data from a file
2. Write our own function allowing us to populate the World with ASCII values from a file containing character data

Since our World stores all values as floating-point numbers, and since character values can be converted to integers which can be converted into floating-point numbers, we know that our ambition is reasonable.

A complete solution for this, called `BoggleWorld`, may be found in Appendix A.

4 Solution

Looking at the API reference for `Solution` shows that it's pretty closely married to the Traveling Salesman problem; feasibility and penalty components make absolutely no sense when talking about `Boggle`. As mentioned in the "Detailed Description" portion of the API reference,

`Solution` is arguably too specialized. Not all `Solutions` will incorporate F and P, after all. However, original attempts to refactor this into a more pleasing shape have been unsuccessful. If you're feeling your oats, it's definitely a worthwhile thing to try.

So it would appear we are, for the moment, stuck with an awkward fit: trying to make `Solutions` optimized for TSPTW and VRPTW work with `Boggle`. The best solution would, of course, be to refactor the code in our `Copious Free Time`⁶. Until then, we will simply ignore the TSPTW and VRPTW components of `Solution`, and add our own functionality as needed.

4.1 Required Methods

There are a few pure virtual methods in `Solution` which must be provided in any instantiable derived class, such as `BoggleSolution`. These are listed in the API reference: `getCost()`, `spawn()`, `computeAll()`, `updateAll()` and `copyFrom()`. Of these, the only two we're really concerned about are `spawn()` and `copyFrom()`; the others are TSPTW or VRPTW-specific, and may be replaced with stubs.

`spawn()` is necessary because an `Engine` may not know precisely what kind of `Solution` it's working on—only that it conforms to the `Solution` class declaration. This lack of knowledge makes it a little difficult for the `Engine` to create new `Solutions`, should those be needed. Thus, our first question is *will our Engine ever need to create new Solution*

⁶From the *Jargon File*: "1. [ironic] A mythical schedule slot for tasks held to be unlikely or impossible. Sometimes used to indicate that the speaker is interested in accomplishing the task, but believes that the opportunity will not arise. 2. [archly] Time reserved for bogus or otherwise idiotic tasks." It is left to the reader to decide in which sense the phrase is here being used.

objects? If so, then we definitely need to implement `spawn()` in such a way that it will return a pointer to a new `BoggleSolution`. If not, then we can replace this with a stub as well. As it turns out, our `BoggleEngine` works by construction and not derivation, so we'll never need to create new solutions.

`copyFrom()` is likewise necessary: if the Engine doesn't know what kind of object is being used, then how can it assign the contents of one to another? By creating a polymorphic `copyFrom()`, it allows us to do the Right Thing⁷ automagically⁸ However, since our Engine is going to work by construction, it will never need to copy Solutions, and this too can be replaced with a stub.

Careful readers will note that two functions described as "necessary" have been handwaved down to stubs. Particularly careful readers will note that five functions described as "necessary" have been handwaved down to stubs. It is only necessary that the functions exist, not that they serve any useful purpose; deciding whether they should do something useful and if so, what, is one of the principal tasks in crafting a Solution.

We also need to write our own custom

⁷Ibid. "That which is *compellingly* the correct or appropriate thing to use, do, say, etc. Often capitalized, always emphasized in speech as though capitalized. Use of this term often implies that in fact reasonable people may disagree. ...Oppose *Wrong Thing*." (emph. in orig.)

⁸Ibid. "Automatically, but in a way that for some reason (typically because it is too complicated, or too ugly, or perhaps even too trivial), the speaker doesn't feel like explaining to you. See *magic*."

`dump()` function, which will be used to send the contents of a `BoggleSolution` to a given `std::ostream`.

Finally, Solutions by default store their internal state (their path through the matrix) as a vector of integers. In the case of TSPTW and/or VRPTW problems, this choice makes a lot of sense, since a solution is just a path from one customer to the next. Where it utterly fails to make sense is for Boggle, where we wish to keep track of a vector of words. Thus, our new `BoggleSolution` will add one additional data member, a `std::vector<std::string>` called `_words`.

5 Engine

Our `BoggleEngine` works via recursive exhaustive search. It needs to keep track of a dictionary of valid English words (which can usually be found on UNIX machines as `/usr/share/dict/words`), and needs to define a function *recurse* which takes as a parameter the following:

1. A matrix, *cover*, of dimension equal to W_b (which we find by going through S_b , representing which cells we've already visited and thus cannot visit again)
2. A string, *word*, containing the word we've found so far

At each step in *recurse*, it does the following:

1. Mark our current cell in *cover*, denoting it as visited
2. Append our current cell–letter from W_b to *word*⁹
3. Do a binary search in our dictionary to find the insertion point P for *word*
 - (a) If what is at P equals *word*, we have found a word. Add *word* to S_b and continue recursion from this point, visiting all the unvisited cells adjacent to us.
 - (b) If *word* is not the prefix of whatever is at P , then we know we’ve constructed an invalid word. Terminate recursion.
 - (c) If *word* is the prefix of whatever is at P , then we know we *may* be on the right track to constructing a word. Recurse from this point, visiting all the unvisited cells adjacent to us.
4. Finally, return.

Please note that step 3(b) is necessary to avoid an explosively bad search strategy. With step 3(b), we prune dead search subtrees as soon as they’re found. Without it, we’d be stuck doing an essentially factorial number of steps to find essentially polynomial numbers of solutions. Djinni is a good

⁹Remember that W_b stores elements as floating-point numbers, so we will have to convert it back to a character value. Also remember that if our cell contains Q, we need to add QU.

tool, but there is no substitute for a well-tuned search algorithm. Believing that efficiency of algorithms is irrelevant in our age of fast computers is living in sin.

6 Runtime

Our Runtime is responsible for assembling together the tuple of $\langle World, Solution, Engine \rangle$. That means loading a BoggleWorld in from disk and creating a BoggleSolution and BoggleEngine as appropriate. Once we know what the Runtime is responsible for doing, we know what our `Parameters.txt` file will contain.

The first line of our `Parameters.txt` is always an identification string. The second line is always the type of problem we’re solving. Past that, the file is dependent upon the type of problem. In the case of Boggle, we really only need one additional line: the name of a file containing our Boggle matrix.

Once we have our BoggleWorld, our BoggleSolution and our BoggleEngine created, making the Runtime work properly is a piece of cake. Open up `Runtime.cc` and alter the two constructors so that they understand the new problem type of “Boggle” and load BoggleWorlds from the specified filename.

7 Putting It All Together

Write your new `Parameters.txt` file.

Edit the Makefile, particularly the targets “all” and “test”, to reflect your changes.

“make test” will compile a purely C++ executable containing your new code. Debug your code using this.

Once your new C++ classes work properly and you’re getting correct solutions, type “make” to rebuild the Djinni scripting language modules.

8 Further Directions

The Solution class is married uncomfortably close to TSPTW and VRPTW problems. This is an artifact of the development process; while one of the goals was to make Djinni extensible to other problems, TSPTW and VRPTW were our biggest priorities. There’s likely a bit of beneficial refactoring which could go on there.

The CompressedAnnealer class takes advantage of the tight coupling to the TSPTW and VRPTW problems. Attempting to use the CompressedAnnealing engine with Solutions which are far outside the TSPTW and/or VRPTW problems may be problematic.

The TSPRoute and CompressedAnnealer classes should probably not be studied to see how Djinni works. They’ve been heavily optimized for performance, which has come at some cost to readability. This is not unexpected: those classes are meant to solve real-world problems, not for pedagogy.

If you’re going to change the code, update the documentation! Good, clear docu-

mentation is something every programmer should demand. Bad or out-of-date documentation is something no programmer should tolerate. Far better there be no documentation at all than significantly out-of-date documentation.

9 A Note On Names

In Semitic mythology and the Islamic faith, a *djinn* is an invisible spirit with free will, capable of both great good and great evil. Iblis (the Islamic name for Satan) was a djinn (c.f. Surat Al-Kahf, 18:50). Both “djinn” and “djinni” are accepted English transliterations of the original Arabic word; however, it was earlier (and widely) transliterated as “genie”.

During the protracted development stage, our software alternated between mocking us and helpfully obeying us. This maddening cycle of Heisenbug¹⁰ and Schrödinbug¹¹ led us to occasional despair. Given the choice of blaming our difficulties on our own programming or on the work of malicious evil spirits, well...certainly, evil spirits have to be at work, right? And so Djinni was so named in honor of the feckless spirit haunting our code.

¹⁰Ibid., “A bug that disappears or alters its behavior when one attempts to probe or isolate it.”

¹¹Ibid., “A design or implementation bug in a program that doesn’t manifest until someone reading source or using the program in an unusual way notices that it never should have worked, at which point the program promptly stops working for everyone until fixed.”