TreeSearch User Guide

Version 0.9

Derrick Stolee
University of Nebraska-Lincoln
s-dstolee1@math.unl.edu

March 24, 2011

Abstract

The TreeSearch library abstracts the structure of a search tree in order to manage jobs in a distributed computation. This document details the strategy of TreeSearch along with implementation, compilation, and execution details. An example application is also given.

1 Introduction

The computation path of a dynamic search frequently takes the form of a rooted tree. One important property of each node in this tree is that the computation at that node depends only on the previous nodes in the ancestral path leading to the root of the computation. If the search is implemented in the usual way, subtrees operate independently.

For a search of this type, all search nodes at a given depth can be generated by iterating through the search tree, but backtracking once the target depth is reached. Each of the subtrees at this depth can be run independently, and hence it is common to run these jobs concurrently (See [2] Chapter 5 for more information).

The TreeSearch library was built to maximize code reuse for these types of search. It abstracts the structure of the tree and the recursive nature of the search into custom components available for implementation by the user. Then, the ability to generate a list of jobs, run individual jobs, and submit the list of jobs to a cluster are available with minimal extra work.

TreeSearch is intended for execution on a distributed machine using Condor [3], a job scheduler that uses idle nodes of a cluster or network. Condor was chosen as its original development was meant for installation in computer labs and office machines at the University of Wisconsin–Madison to utilize idle computers.

The C++ portion of TreeSearch is independent of Condor. The Python scripts which manage the input and output files as well as modifying the submission script are tied to Condor, but could be adapted for use in other schedulers.

1.1 Acquiring TreeSearch

The latest version of TreeSearch and its documentation is publicly available on GitHub [1] at the address http://www.github.com/derrickstolee/TreeSearch/.

2 Strategy

Let us begin by describing the general structure and process of an abstract tree-based search. There is a unique root node at depth zero. Each node in the tree searches in a depth-first, recursive manner. There are

a number of children to select at each node. One may select this child through iteration or selecting via a numerical order. Before searching below the child, a pruning procedure may be called to attempt to rule out the possibility of a solution below that child. Another procedure may be used to find if this node is a solution. Now, the search recurses at this node until its children are exhausted and the search continues back to its parent.

2.1 Subtrees as Jobs

This tree structure allows for search nodes to be described via the list of children taken at each node. Typically, the breadth of the search will be small and these descriptions take very little space. This allows for a method of describing a search node independently of what data is actually stored by the specific search application. Moreover, the application may require visiting the ancestor search nodes in order to have consistent data. With the assumption that each subtree is computationally independent of other subtrees at the same level, one can run each subtree in a different process in order to achieve parallelization. These path descriptions make up the input for the specific processes in this scheme.

Each path to a search node qualifies as a single job, where the goal is to expand the entire search tree below that node. A collection of nodes where no pair are in an ancestor-descendant relationship qualifies as a list of independent jobs. Recognizing that the amount of computation required to expand the subtree at a node is not always a uniform value, TreeSearch allows a maximum amount of time within a given job. In order to recover the state of the search when the execution times out, the concept of *partial jobs* was defined. A partial job describes the path from the root to the current search node. In addition, it describes which node in this path is the original job node. The goal of a partial job is to expand the remaining nodes in the subtree of the job node, without expanding any nodes to the left of the last node in this path. See Figure 1 to an example partial job and its position in the job subtree.

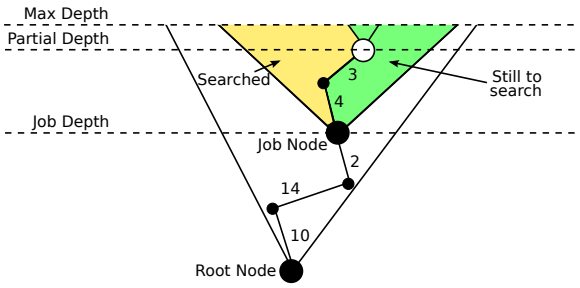

Figure 1: A partial job description.

2.2 Job Descriptions

The descriptions of jobs and partial jobs are described using text files in order to minimize the I/O constraints on the distributed system. The first is the standard job, given by a line starting with the letter `J`. Following this letter are a sequence of numbers in hexadecimal. The first two should be the same, corresponding to the depth of the node. The remaining numbers correspond to the child values at each depth from the root to the job node.

A partial job is described by the letter `P`. Here, the format is the same as a standard job except the first number describes the depth of the job node and the second number corresponds to the depth of the current node. For example, the job and partial job given in Figure 1 are described by the strings below:

```
J 3 3 10 14 2
P 3 5 10 14 2 4 3
```

2.3 Customization

The TreeSearch library consists of an iterative implementation of the abstract search. The corresponding actions for a specific application are contacted via extending the `SearchManager` class and implementing certain virtual functions. The list of functions available are given in Table 1.

`LONG_T`	`pushNext()`	Deepen the search to the next child of the current node.
`LONG_T`	`pushTo(LONG_T child)`	Deepen the search to the specified child of the current node.
`LONG_T`	`pop()`	Remove the current node and move up the tree.
`int`	`prune()`	Perform a check to see if this node should be pruned.
`int`	`isSolution()`	Perform a check to see if a solution exists at this point.
`char*`	`writeSolution()`	Create a buffer that contains a description of the solution.
`char*`	`writeStatistics()`	Create a buffer that contains custom statistics.

Table 1: List of virtual functions in the `SearchManager` class.

In addition to supplying the logic behind these functions, protected members of the `SearchManager` class can be modified to change the operation of the search. These parameters are listed in Table 2.

Type	Name	Option	Description
`int`	`maxdepth`	`-m [N]`	The maximum depth the search will go. In generate mode, a job will be output with job description given by the current node.
`int`	`killtime`	`-k [N]`	Number of seconds before the search is halted. If the search has not halted naturally, a partial job is output at the current node.
`int`	`maxSolutions`	`--maxsols [N]`	The maximum number of solutions to output. When this number of solutions is reached, a partial job is output and the search halts.
`int`	`maxJobs`	`--maxjobs [N]`	The maximum number of jobs to output (generate mode). When this number of jobs is reached, a partial job is output and the search halts.
`bool`	`haltAtSolutions`	`--haltatsols [yes/no]`	If `true`, the search will stop deepening if `isSolution()` signals a solution. If `false`, the search will continue until specified by `prune()` or `maxdepth`.

Table 2: List of members in the `SearchManager` class.

3 Integration with TreeSearch

This section details the specific interfaces for implementation with TreeSearch.

It is important to understand the order of events when the search is executing. The search begins when the `doSearch()` methd is called. The first call initializes the search, including starting the kill timer. Then, each recursive call expands the current search node at the top of the stack. Figure 3 describes the actions taken by the recursive `doSearch()` method takes at each search node.

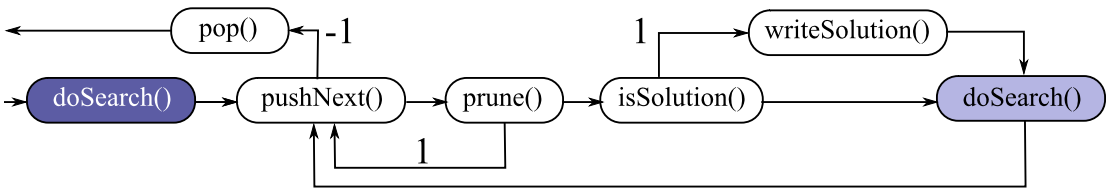

Figure 2: The conceptual operation of the `doSearch()` method.

3.1 Virtual Functions

The two most important methods are the `pushNext()` and `pushTo(LONG_T child)` methods. Both deepen the search, manage the stack, and control the job descriptions. Each returns a child description (of type `LONG_T`)

`pushNext():` Advance the search stack using the next augmentation available at the current node. Return a label of `LONG_T` type which describes the augmentation. Return -1 if there are no more augmentations to attempt at this stage, signaling a `pop()` operation.

`pushTo(LONG_T child):` Advance the search stack using the augmentation specified by `child`. If the augmentation fails as specified, return -1, and the search will terminate in error. **Note:** If this method is called during the partial portion of a job description, the later augmentations will be called using the `pushNext()` method, so the current augmentation must be stored for later.

`pop():` Clean up any memory used for the current level and/or revert any data structures to the previous level.

`prune():` Check if the current node should be pruned (i.e. detect if there are no solutions reachable from the current node by performing the specified augmentation). Return 1 if the node should be pruned, 0 otherwise. A prune signal will be followed by the `pop()` method.

`isSolution():` Check if there is a solution at this node. If there is a solution, store the solution data and return 1. The `writeSolution()` method will be called to pass the solution data to output. If the `haltAtSolutions` option is set, a solution will trigger a `pop()` method. Otherwise, the node will be used for augmentations until the maximum depth is reached or the `prune()` method signals a prune for all later nodes.

`writeSolution():` Return a buffer containing the solution data for the output. This data will be sent to standard out along with the job description of the current node (with an '`S`' prefix). If your data is prohibitively large for sending over the network, this job description can be used to generate the current node where the solution data can be recovered. **Note:** The buffer must be allocated using `malloc`, as it will be deallocated using `free`.

`writeStatistics():` Write a buffer of custom statistics information. Each line must follow the format

T [TYPE] [ID] [VALUE]

where [TYPE] is one of "SUM", "MAX", or "MIN", [ID] is a specified name, and [VALUE] is a number. These statistics are combined by the `compactjobs` script using the [TYPE] specifier (either sum the values or store the maximum or minimum) and the statistics are placed in the `allstats.txt` file. The `combinestats` script converts the `allstats.txt` file into a comma separated value file, grouping by depth over variables with [ID] of the form [SUBID]_AT_[#]. This allows per-depth statistics tracking.

3.2 Helper Methods

The following methods are useful when constructing a TreeSearch application.

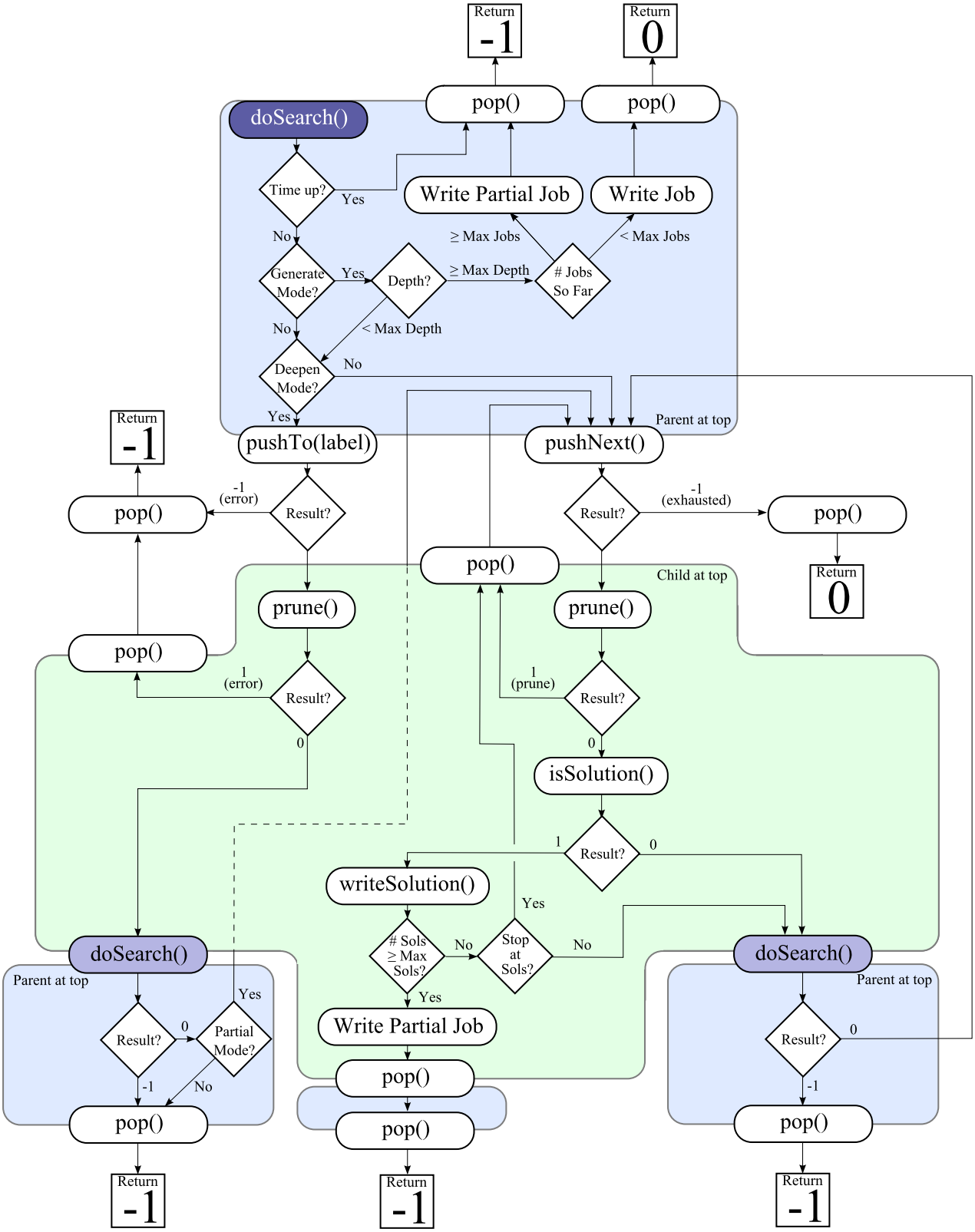

Figure 3: The full operation of the `doSearch()` method.

`importArguments(int argc, char** argv):` Take the command line arguments and set the standard search options. The options from Table 2 such as `killtime`, `maxJobs`, and `maxdepth` are all set in this way.
`readJob(FILE* file):` Read a job from the given input source, such as standard in. It will read only one line, and prepare the manager for the `doSearch()` method.
`doSearch():` This method starts the search on the current job as well as returns the status of the search: 1 if completed with a solution, 0 if completed with no solution, and -1 if halted early due to error or time constraints.

3.3 Compilation

To compile TreeSearch, run `make` in the source directory. This command compiles the object file `SearchManager.o` which must be linked into your executable. Moreover, it compiles the example application presented in Section 5. Your code must reference the header file `SearchManager.hpp` and link the object file `SearchManager.o`.

4 Execution and Job Management

To execute a single process, simply run your executable with the proper arguments. However, to run a distributed job via Condor, a set of scripts were created to manage the input and output files, the Condor submission file, and monitor the progress of the submission during execution.

4.1 Management Scripts

The TreeSearch library works best with independent subtrees and hence does not suffer from scaling issues when the parallelism is increased. However, managing these large lists of jobs requires automation.

4.1.1 Expanding jobs before a run

When the generation step is run, a list of jobs is presented in a single file. Condor requires a separate input and output file for each process. The role of the `expandjobs` script is to split the jobs into individual files and to set up the Condor submission file for the number of jobs that are found.

There are a few customizable options for this script.

- `-f [folder]` – change the folder where the jobs are created. Default is `./`.
- `-m [maximum]` – set the maximum number of jobs allowed. Default is unlimited.
- `-g [groupsize]` – set the number of jobs per process. Default is 1.

Inside the specified folder, the file `condorsubmit.sub.tmp` is modified and copied to `condorsubmit.sub` with the proper queue size based on the number of jobs found. Any remaining jobs that did not fit within the maximum are held as back jobs. They will be added to the job pool when the jobs are completed.

4.1.2 Collecting data after a run

Once Condor has completed the requested jobs, the output must be collected to discover which jobs completed fully, which are partially complete, and how many solutions have been found. The script `compactjobs` was built for this purpose.

This script takes the output files and reads all new jobs that may have been generated using the staging feature, finds if the input job completed or is partial, and reports on the total number of jobs of each type. Moreover, it will find and store the solutions found, along with the corresponding data.

Finally, it compiles statistics from each run. Using the `writeStatistics` method, the application may report statistics by starting the line with a "T" followed by the type (MAX, MIN, SUM), variable name, and variable value. These are collected using the specified type and compiled with existing statistics from previous batches.

5 Example Application

An example application is given in the file `example.cpp`. This example application takes an extra option `--bits [k]`, where k is an integer no more than the maximum depth specified (m). The solutions are the incidence vectors for all subsets of $[m]$ which have exactly k elements. The augmentation procedure places a 0 or 1 in the next bit of the incidence vector, corresponding to the choice of placing d in the set (0 it is out, 1 it is in) where d is the current depth. The `prune()` method prunes when there are more than k elements selected.

This example should highlight a few nuances when working with the TreeSearch library, specifically how the root node is used to hold the latest label of the first node, and how the `SearchNode` class is extended to include necessary information for the current node so that the `prune()` and `isSolution()` methods do not need to access more than one position in the stack.

6 Example Workflow

When managing a distributed search, there are several choices to make as the user. What depth should I generate to? How many jobs should I run? How long should I set the kill time? These questions are answered based on your application and experience. This section guides you through the use of the management scripts as well as strategies for different situations.

6.1 Create the Submission Template

The file `condorsubmit.sub` contains the necessary information for submission to the Condor scheduler. The number of processes is automatically managed by the `expandjobs` script. However, you may modify the arguments of the run depending on the type of job you want to run. More on this later.

6.2 Generate initial jobs

To create a beginning list of jobs, run a single process in `generate` mode with a reasonably small maximum depth. Send the output to a file called "`out.0`" in order to have it viewed by the `compactjobs` script. To start at the root search node, use the job description "`J 0 0`". The jobs created could be reasonably small.

6.3 Compact data

After any run, use the `compactjobs` script to combine the list of results, partial jobs, and new jobs from the output files into a collection of data files. This will also give you the number of jobs which completed, failed, are partial, or are new.

6.4 Evaluate Number of Jobs

Based on this number of jobs, you have a few options.

1. You have a lot of jobs ($\geq 20,000$ for instance). This is probably too many to run each job as a single process, so they must be grouped together. When using the `expandjobs` script, use the `-g` flag to group jobs together into processes, so that there are a reasonable number of total processes. For example, if there were $100,000$ jobs, using `expandjobs -g 10` would result in a list of $10,000$ processes. After running `expandjobs`, modify the generated `condorsubmit.sub` script to set the killtime so that all of the jobs in the process can complete. For example, if I want to run the previous list of $10,000$ processes for an hour each, I want to set the per-job killtime to six minutes, or 360 seconds.

   Hopefully, many of your processes will complete in this short time interval, and you can run the remaining processes for a longer period. If this does not occur, and you still have many jobs remaining, perhaps using the `-m` flag on the `expandjobs` script to bound the total number of jobs will allow fewer jobs per process while storing the remaining jobs for execution later.

2. You have a decent number of jobs (between $2,500$ and $20,000$). This is a good number for running each as an individual process. Running `expandjobs` with no grouping will allow a bijection between processes and jobs. Modifying the generated `condorsubmit.sub` file for a per-process killtime of one hour (3600 seconds) will allow a reasonable amount of computation per job.

3. You have very few jobs (below $2,500$) which did not terminate in an hour of computation. With the number of jobs, just repeating another hour-per-job submission will not take full advantage of parallelism. Use the `expandjobs` script (possibly with grouping) to generate a list of input files for your jobs. Modify the `condorsubmit.sub` script to be in `generate` mode with a maximum depth beyond your current job-depth. Depending on the density of your application's branching, this could be between one to ten or more levels beyond the current job depth. It is usually a good goal to create a large number ($\geq 20,000$) of jobs which can be run with grouping at a small computation time in order to quickly remove small subtrees of the search. This hopefully isolates the "hard cases" that kept the current list of jobs from completing.

6.5 Submit Script

Using the `condor_submit` script, submit the processes using the `condorsubmit.sub` submission script. Now, wait for the processes to complete. You can monitor progress by using two Condor commands:

1. `condor_status -submitters` will give you a list of running/idle/held jobs for each submitter. This is a good way to watch your queue when many jobs are running.

2. `condor_q [-submitter username]` or `condor_q [clusterid]` will return a per-process list of run times, statuses, and other useful information. This is not recommended when there are many processes running, but when there are less than 100 processes, this can help to find processes that are not completing or when you should expect the processes to complete. Use the `-currentrun` flag to see how long the processes have run since their last eviction or suspension.

If the above methods are not telling you the information you want, view the tail of the log file (as specified in your submission script). This contains the most up-to-date information including the amount of memory each process is using and the reasons for evictions or other failures.

If your processes are not completing, or you have found that you incorrectly set the submission script, remove the jobs using the `condor_rm [clusterid]` command.

7 Summary

You should now have the necessary information to develop your own applications using the TreeSearch library. For support questions or bug reports, please email the author at s-dstolee1@math.unl.edu.

8 Acknowledgements

This software was developed with the support of the National Science Foundation grant DMS-0914815 under the advisement of Stephen G. Hartke.

The author thanks the faculty and staff at the Holland Computing Center, especially David Swanson, Brian Bockleman, and Derek Weitzel for their extremely helpful advice during the design and development of this library.

References

[1] GitHub Inc. Github. https://github.com/.

[2] Petteri Kaski and Patric R. J. Östergård. *Classification Algorithms for Codes and Designs*. Number 15 in Algorithms and Computation in Mathematics. Springer-Verlag, Berlin Heidelberg, 2006.

[3] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.