Figuring out why a process in a batch window ended late often requires tracing back through the critical path to look for anomalies. In a complicated batch schedule, determining which jobs are on the critical path can be a seemingly daunting task. Various commercial products can be used for this task, but they may not be necessary. The code needed to determine which predecessor jobs constitute a particular job’s critical path is relatively, perhaps surprisingly, simple.

Introduction

The term “critical path” is used in a couple of different ways. The loose connotation is that the critical path is the list of things that must get done before an entire project is complete. In this form, the critical path is little more than a list of items to be completed. In project management, this is sometimes limited to just the items that take the longest.

When performance analysts talk about the critical path they are most often referring to the path through a network of interrelated batch jobs that determines the overall elapsed time of the complete process. Determining this critical path is often seen as a difficult challenge for processes that involve hundreds or even thousands of inter-related jobs. I think this perception comes from our tendency to want to start at the first job and map out all the interdependencies and evaluate all the possible path lengths. Indeed, traditional project management critical path analysis works this way. However, I have found that when I have been interested in the critical path, the question is a simple one, one that has a simpler solution: What series of events caused this job to end late?

Terminology

To avoid any possible confusion, I will define a few terms. A “batch schedule” is the list of all the batch jobs that must be executed in a given time period, which is often referred to as the “batch window”. Jobs in the schedule have predecessor-successor relationships; some jobs must complete before certain other jobs can start. The processing of the various jobs in the schedule may be referred to as the “batch flow”. The relationships between the jobs may be represented as a chain:

```
  JobA  JobB  JobC  JobD
   19:00

In this case, JobA is the predecessor to both JobB and JobC, both of which are predecessors to JobD. Or you could say that JobD is the successor to JobB and JobC, which are both successors to JobA.

In addition to the dependencies on other jobs in the schedule, a job may have time dependencies as well. For example, perhaps JobC is not allowed to start until 19:00. So even if JobA finishes at 18:30, JobC must wait another half hour before starting.

Complications

Clearly this is a trivial example—in the real world batch schedules with hundreds or thousands of jobs with much more complicated relationships are the norm. There may also be extraneous relationships within the schedule. For example, if the schedule indicated JobA is a predecessor to JobD, that would be an unnecessary relationship because all of JobD’s predecessors already have JobA as a predecessor. Such an extraneous relationship has no impact on the batch flow but may complicate your analysis efforts.
For large applications, the schedule often changes day to day based on business process schedules. A payroll extract process may be inserted bi-weekly. A general ledger export may happen once a month. Balancing reports may run on the first workday of the month. Such periodic jobs may be inserted anywhere in the chain of jobs, so you can not assume that the schedule for tonight's batch window will be the same for tomorrow or next Monday.

If you are looking at an actual execution of a batch window you may find that some jobs failed and had to be re-run. Such issues may cause significant changes to the critical path.

So unless the application being studied has a trivial number of jobs and a relatively static schedule, determining the critical path likely will require the use of some sort of automated tool.

**Tool candidates**

There are tools that integrate with the various batch scheduling products to do critical path analysis. I have not used these tools, but the vendors' advertisements certainly make them appear to be useful. Of course those tools typically have a cost associated with them that has to be justified, which seems to be difficult for many shops.

There are also more generic tools that perhaps could be employed as well. For example, project scheduling tools (such as Microsoft Project) generally have the ability to take a list of tasks with dependencies and produce Gantt or PERT charts which can be used to visually determine the critical path. However, such products are not really designed to deal with a batch window and so may have difficulty dealing with tasks that last for seconds or minutes. Furthermore, getting the data from your batch scheduler product into such a product may be problematic as well. While some have used such products successfully [SCH99], I have found it to be problematic, although there is at least one product that purports to make the process easier. (See reference in [ZAS01].)

SAS has a product (SAS/OR) that reportedly does critical path analysis as well. Again, I do not have experience with this tool, but it is probably worth exploring if you have funds available to purchase a tool.

Of course, I am sure that I am not the only performance analyst who routinely lacks funding to purchase new tools, no matter how useful (and in some cases how inexpensive) they might be. So the remaining option is to create your own tool. The rest of this paper will describe such a tool that I built to satisfy most of my critical path analysis needs.

**The Question**

The perceived difficulty in finding the critical path generally seems to have two causes:

1) Looking at the problem from the perspective of the entire schedule, usually starting at the first job to be executed.

2) Trying to evaluate the schedule based on elapsed times.

These are valid constraints if you are trying to predict the duration of an upcoming project. But they are immaterial for the question I find myself most often needing an answer to, Why did we miss this batch SLA target last night? We have various SLAs (Service Level Agreements) which specify that various jobs should be complete by a particular time of the night. For example, perhaps customer bills must be sent to the printer by 02:00. So we record the end time of the job that prints the bills and if it completes after 02:00, people want to know why. In some cases the answer is obvious (e.g., “the main billing routine had an error and it took the programmer 2 hours to find and fix it”), but in other cases the problem is a more subtle issue with multiple jobs in the flow. The first step in determining an answer in this case is finding the critical path and then comparing the execution of the jobs on the critical path to their historic norms. But first you have to have the critical path.

**The Answer**

For this scenario, determining the critical path is not difficult. All you need is the job schedule with the job dependencies and the job execution details for the batch window execution in question. In particular you need the predecessor relationships and the job end times.

With these in hand, it is very easy to find the critical path. Start with the final job that you are interested in: the bill print job that finished at 02:45 instead of 01:45. For that job, look at all its predecessors. Find the one with the latest end time. That by definition is the tail end of the critical path leading to the bill print job. Now repeat the process for that job, building the critical path backwards. You may arbitrarily decide to stop building the critical path after a certain number of iterations or when you reach a particular job. Or when the only predecessor for the critical job is a time of day, you have taken the analysis as far as you can.

This is relatively easy to do by hand if the batch schedule is not too large, but it is also relatively easy
to do programmatically. If you write a program to do it for you, that program could also do other interesting things like displaying the critical path graphically or integrating historical data. Such a utility will greatly enhance your ability to quickly answer that seemingly difficult question of why that bill print job was 45 minutes late last night!

Building the Process

It is not uncommon to need to review the performance of a batch window from a few days ago, so it is necessary to have historical data. I already had historical batch job performance data stored in a performance database which was also externalized in XML files. Because the batch schedule changes daily, I added a new process to capture the schedule data on a daily basis as well. I did this by running a report that printed the entire schedule and then parsed that report for the job relationships and stored the result in an XML file. A small example portion of this file is shown in Appendix A.

Note that in the scheduler we use, the schedule is broken into “applications” and each batch job is an “operation” within an application. In the XML file, both the predecessors and successors to each job are stored as children of the job in `<pred>` and `<succ>` elements. Within those elements, the related job is indicated by the job’s application id and operation id. Where there is a time dependency, the application id is stored as “Time” and the operation id is the actual time that the job must start after.

I create one of these schedule XML files each day. The file name includes the date the data is for. A separate XML file lists the XML schedule files that are currently available.

Explaining the Code (See Appendix B)

Having captured the schedule, I then built a JavaScript application that runs in the browser to help with my analysis. The program loads the schedule and job history XML files, then lets the user select a batch date and batch job to analyze. It builds the critical path backwards from that job and displays it graphically. An example is shown above. Since this paper is about finding the critical path, I will not complicate the topic by discussing the code that does the graphical display. If you can understand the process to build the critical path, you can display the results however you want. A simple textual report would likely be sufficient for a lot of problems.

The crux of the JavaScript code that determines the critical path is shown in Appendix B. The remainder of this paper will discuss, and hopefully explain, that code. It should not matter much if you have not previously worked with JavaScript—it’s a procedural language like many others. See the references at the end of the paper for a reference guide. It may be helpful to note that array subscripts are denoted with square brackets and that properties of an object are
denoted by placing a period between the object and element. So if we had an array of job objects, each with a name and a description, the n° job would be referenced like this:

\[
\text{job}[n].\text{name} \\
\text{job}[n].\text{desc}
\]

In my application, I also do some forward-looking at the successor jobs, primarily because I am sometimes interested in the immediate successor to a job. However, that code does not truly find the critical path, so is not included in Appendix B.

There are two common terms used in the code shown: levels and chains. Levels refer to how many steps away from the job under study we are. The jobs that are immediate predecessors to the initial job are at level -1, the predecessors of those jobs are at level -2, etc. The successors to the initial job would be at level +1. The chain is not actually a linked list of all the jobs but merely an array of all the jobs that have been found and placed in a level. Building a true linked list of the schedule seems like a logical thing, but because each job may have multiple predecessors and successors, the schedule is really more of a mesh than a chain. In relation to the chain, entries for jobs are called links.

The process begins in function buildChain, which takes as input a particular application and operation to use as the initial job to calculate the critical path. The job name is also passed in just so this function does not have to look it up—there is a one-to-one relationship between application/operation and job name. The combination of application (“app”) and operation (“op”) is referred to as “appop” in the code and is used as an index into the chain.

Some specific code to load the schedule for the particular batch date under study is omitted from Appendix B for clarity. Continuing with the code, arrays get initialized and a new level 0 added. We then loop for as many levels as was requested (pLevels, a global variable set elsewhere), starting at level 0. At each level, we add all the predecessors for all the jobs at that level by calling the addPreds function. We then record the critical path for that level by calling the recordCP function.

The addPreds function is shown in Appendix B immediately after buildChain. It loops through all the jobs on the chain (the k loop) and looks for ones that exist at the current level. For those jobs, it loops through all the job’s predecessor jobs (the j loop) and adds those jobs to the chain at the predecessor level to the current level by calling the addLink function.

The addLink function is shown next in Appendix B. If the current application/operation (that is the job) does not currently exist on the chain, then it is added to the chain. The job is added as an object, which includes attributes such as the level the job is currently at, the identifying information, and an array of executions (since the job may have had to have been re-run multiple times). Before leaving addLink, the maximum end time for the level that the job was added to is also updated.

If the job already existed on the chain, its level is merely updated to the new level. You might think that would not be necessary, but that handles the case where the schedule contains logically unnecessary relationships. For example, in the following diagram, A does not need be a predecessor to D, but if it is recorded as such, then it needs to be recorded at level -2, not -1.

\[
\text{Level -2} \quad \text{Level -1} \quad \text{Level 0}
\]

- Level -2
- Level -1
- Level 0

A ——— B

C ——— D

Updating the Critical Path

The recordCP function updates the critical path for a given level. The critical job for each level is identified by the cpApp and cpOp variables in the dbLevel array. For level 0, we simply record the times from the last run for the job at level 0. For the other (negative) levels, we find the job that was the critical path on the level that is one higher (plvl) than the level (lvl) we’re working on. So if we are working on level -2, we use the critical job from level -1. We then loop through all the predecessors of that job (the k loop), and all the runs of each predecessor (the r loop). If a run’s end time is later than the current maximum end time for the level, we update the maximum end time and record the critical path as being that job. When the loops end, dbLevel[lvl].cpApp and .cpOp will have the correct critical job for that level.

Note the empty level skip check in the code—again that’s only in there because the schedule could have
illogical, but technically valid, relationships in it. Additionally, because we are only looking at a subset of the schedule, some of the relationships may make sense in terms of the overall schedule, but have no bearing on the critical path flow. So while it does not seem like you should need this check, I found I did need it for the production schedule I most often investigate.

**Reading and Parsing XML**

The XML file for the schedule is very large, so I only parse when needed. The parseOPC function shown in Appendix C adds an application to the dbApps array and is called when an application is found that does not currently exist in dbApps. While this is not directly related to finding the critical path, a bit of explanation of the methodology may be useful if you are not familiar with parsing XML files. It will be helpful to also refer to the XML snippet shown in Appendix A.

The methodology used here to parse the XML is to use the DOM (Document Object Model) functions built into JavaScript. These DOM functions can also be used to parse and manipulate HTML—which is also quite useful but beyond the scope of this paper.

The function starts by selecting the app node that contains the attribute "id" that is equal to the application that we want to parse and add to the array. This is a simple XPath expression and is much faster than iterating over the entire file looking for the application we are interested in. The function returns an array of nodes. Although we expect to get back only one entry, to be safe we do process all members of the array.

The getAttribute function is used to return the value of an attribute of an element, for example the value of the "id" attribute. Note that the getAttribute function is applied to an entry from the array returned by the XPath expression that selected app elements, so this is the id of our application. (Using the XML from Appendix A, this would be “#AMCSPREBATCH”.)

The getElementsByTagName function returns an array of all the elements of a given name. In this case we get an array of all the operations which are represented by the “op” elements. We then iterate over that array, pulling some attribute values and a few element values. For example, the "wkstn" values for the operations shown in Appendix A are all “CPUJ”. We similarly iterate over arrays for the predecessor and successor elements and pull data from those nodes.

While traversing the XML document via the DOM functions is not particularly difficult, it is not nearly as quick (in terms of code or elapsed time) as simply doing an array lookup, which is why we extract the data into arrays for use elsewhere in the code.

**Summary and Other Ideas**

Critical path analysis, applied to answer the common performance question, Why did the batch schedule finish late? does not have to be difficult. You simply need the job relationships and the end times for the jobs and then work backwards from the last job.

Some may suggest that this is merely looking back at history and a predictive approach would be more useful. Unfortunately, job execution times can vary for multiple reasons from night to night and schedules change from day to day, so the critical path may also change daily. However the technique described here could be used in a predictive fashion if you fed it predicted job completion times instead of actual completion times. Similarly, if you have multiple “last” jobs that you are interested in, you could (artificially in the input data or in the real schedule) add a common successor to all the jobs you are interested in, then analyze that common job. In reality, we have just such jobs throughout our schedule to record batch SLA attainment.

Hopefully this paper has given you some ideas, and even some example code, that you will find useful in analyzing your own batch schedules.

**References**


Critical Path Analysis for Project Management
http://www.netmba.com/operations/project/cpm/

JavaScript reference
http://www.w3schools.com/jsref/default.asp

W3C XPath reference
http://www.w3.org/TR/xpath

XPath tutorial
http://www.w3schools.com/Xpath/
APPENDIX A – SAMPLE PORTION OF BATCH SCHEDULE XML FILE

```xml
<app id='#AMCSPREBATCH'>
  <op id='3' job='#AMCS106' arr='1730'><wkstn>CPUJ</wkstn><desc>Prt/Audit Trail Entry</desc>
    <pred><aid>Time</aid><opid>1730</opid></pred>
    <succ><aid>#AMCSPREBATCH</aid><opid>6</opid></succ>
  </op>
  <op id='6' job='#AMCS105' arr='1730'><wkstn>CPUJ</wkstn><desc>Screen FICHE Generate</desc>
    <pred><aid>#AMCSPREBATCH</aid><opid>3</opid></pred>
    <succ><aid>#AMCSPREBATCH</aid><opid>9</opid></succ>
  </op>
  <op id='9' job='#ADRM101' arr='1800'><wkstn>CPUJ</wkstn><desc>Apco Quiesce</desc>
    <pred><aid>#AMCSPREBATCH</aid><opid>6</opid></pred>
    <pred><aid>#SMCSPREBATCH</aid><opid>6</opid></pred>
    <succ><aid>#AMCSBACKUPITRON</aid><opid>6</opid></succ>
    <succ><aid>#AMCSFNETPOSTEST</aid><opid>15</opid></succ>
    <succ><aid>#SMCSOTC</aid><opid>3</opid></succ>
    <succ><aid>#AMCSPOSTCASH</aid><opid>3</opid></succ>
  </op>
</app>
```

APPENDIX B – JAVASCRIPT CODE TO FIND CRITICAL PATH

```javascript
// Build a chain of the job dependencies for the current app and operation
function buildChain(curapp, curop, curjob) {
  var appop, curappop;
  var nodes = new Array;
  var runs = new Array;
  var k;
  var initTS;

  dbChain = new Array;
  dbLevel = new Array;

  // Code to find the batch date from the user interface
  // and load the OPC Schedule file omitted
  dbLevelNew(0);
  addLink(0,curapp,curop,curjob);
  for (k=0;k<levels;k++) {
    dbLevelNew(k-1.0);
    addPreds(k);
    recordCP(k); // starts with 0
  }
  recordCP(k); // get the CP for last level

  // forward (successor code omitted—not really the critical path)
  maxEtAll = 0;
  minInitAll = new Date(2099,11,31,23,59,59);
  maxEndAll = new Date(0);
  minLevel = 999;
  maxLevel = -999;
  for (k in dbLevel) {  // determine time period for display purposes
    if ( dbLevel[k].maxEt > maxEtAll) { maxEtAll = dbLevel[k].maxEt; }
    if ( dbLevel[k].minInit < minInitAll) { minInitAll = dbLevel[k].minInit; }
    if ( dbLevel[k].maxEnd > maxEndAll) { maxEndAll = dbLevel[k].maxEnd; }
    if (parseInt(k) < parseInt(minLevel)) { minLevel = k; }
    if (parseInt(k) > parseInt(maxLevel)) { maxLevel = k; }
  }
}
```
// Add predecessors to all the jobs at a given level in the chain

function addPreds(lvl) {
    var capp, cop, cjob, thisop, j, tLink;
    var chainLen = dbChain.length; // chain length will change!
    var newLvl = lvl - 1; // target level we're building

    for (var k=0;k<chainLen;k++) {
        if (dbChain[k].level == lvl) {
            tLink = dbChain[k]; // this link
            tLink.preds = new Array;
            thisop = dbApps[tLink.app].ops[tLink.op];
            for (j=0;j<thisop.pred.length;j++) { // for all preds
                capp = thisop.pred[j].appid; // get pred app
                if (capp != "Time") {
                    checkAddApp(capp); // add app to dbApps if not there
                    cop = thisop.pred[j].opid; // get pred op
                    cjob = dbApps[capp].ops[cop].job; // get pred jobname
                    tLink.preds.push(capp + "_" + cop); // add link to chain
                } else { // this already exists in the chain
                    cop = thisop.pred[j].opid; // get pred op (Time)
                    tLink.preds.push(capp + " " + cop); // include times on links
                }
            }
        }
    }
}

// build a "link" for the chain

function addLink(lvl, curapp, curop, curjob) {
    var tChain;
    checkAddApp(curapp);
    var curappop = curapp + "_" + curop;
    if (! (curappop in dbChain)) {
        var l = dbChain.length;
        dbChain[l] = new Object;
        dbChain[curappop] = dbChain[l];
        tChain = dbChain[l];
        tChain.level = lvl;
        tChain.app = curapp;
        tChain.op = curop;
        tChain.job = curjob;
        tChain.runs = getRunsDate(curjob, batchDate);
    } else { // this already exists in the chain--just update level
        tChain = dbChain[curappop];
        tChain.level = lvl;
    }

    for (var k=0;k<tChain.runs.length;k++) { // update dbLevels here to avoid unneccessarily passing lvl
        if (tChain.runs[k].et > dbLevel[lvl].maxEt) {
            dbLevel[lvl].maxEt = tChain.runs[k].et;
        } else if (tChain.runs[k].init.getTime() < dbLevel[lvl].minInit.getTime()) {
            dbLevel[lvl].minInit = tChain.runs[k].init;
        } else if (tChain.runs[k].end.getTime() > dbLevel[lvl].maxEnd.getTime()) {
            dbLevel[lvl].maxEnd = tChain.runs[k].end;
        }
    }
}
// determine the critical path job for a given level
function recordCP(lvl) {
    var maxEnd, cpAppOp, tchain, plvl, pchain;
    var appOps = new Array;
    maxEnd = new Date(1970,0,1,0,0,0);

    if (lvl == 0) { // critical path for level 0 unique
        for (k=0;k<dbChain.length;k++) { // each job in the chain
            tchain = dbChain[k];
            if (tchain.level == lvl) {
                for (r=0;r<tchain.runs.length;r++) {
                    if (tchain.runs[r].end.getTime() > maxEnd.getTime()) {
                        maxEnd = tchain.runs[r].end; // found critical job at this level
                        dbLevel[lvl].cpApp = tchain.app;
                        dbLevel[lvl].cpOp = tchain.op;
                    }
                }
            }
        }
    } else {
        // Successor search not shown since not critical path
        plvl = lvl + 1; // plvl is the level to look at
        while (dbLevel[plvl].maxEt == 0) {plvl = plvl + 1;} ; // skip empty levels
        cpAppOp = dbLevel[plvl].cpApp +"_"+dbLevel[plvl].cpOp; // prev CP job
        tchain = dbChain[cpAppOp];
        if ((typeof(tchain) != "undefined") && (typeof(tchain.preds) != "undefined")) {
            appOps = tchain.preds; // what to look through
            for (k=0;k<appOps.length;k++) { // for each job to look at
                pchain = dbChain[appOps[k]];
                for (r=0;r<pchain.runs.length;r++) { // look through each run
                    if (pchain.runs[r].end.getTime() > maxEnd.getTime()) {
                        maxEnd = pchain.runs[r].end; // found critical job at this level
                        dbLevel[lvl].cpApp = pchain.app;
                        dbLevel[lvl].cpOp = pchain.op;
                    }
                }
            }
        }
    }
}

// add a dummy entry to dbLevel
function dbLevelNew(lvl) {
    dbLevel[lvl] = new Object;
    dbLevel[lvl].maxEt = 0;
    dbLevel[lvl].minInit = new Date(2099,11,31,23,59,59);
    dbLevel[lvl].maxEnd = new Date(0);
}

// build an array of run execution data for a job / batch date combo
function getRunsDate(curjob, bdate) {
    // This function not shown for clarity, but returns an array of job
    // executions for this job and date. Each entry contains interesting data,
    // including the time the job started (init) and the time it ended (end)
    return runs;
}
// function to see if we already have an app in dbApps, and if not add it
function checkAddApp(app) {
  if (!app in dbApps)) {
    parseOPC(app); // add this app to dbApps
  }
}
APPENDIX C – JAVASCRIPT CODE TO PARSE THE SCHEDULE XML

// parse the OPC XML file into dbApps
// dbApps[].name
// .ops[].id
// .job
// .arr
// .wkstn
// .desc
// .pred[]
// .succ[]
function parseOPC(addapp) {
    var t1 = new Date();
    var o, r, opid, appop, refs;

    // use xpath expression to search the opc plan to find the app we want to add
    apps = xmlOPC.selectNodes("//opc/app[starts-with(@id,'" +
        addapp.toUpperCase()+'')]"));
    for (var a=0;a<apps.length;a++) {
        dbApps[a] = new Object;
        dbApps[a].name = apps[a].getAttribute("id");
        dbApps[dbApps[a].name] = dbApps[a];
        dbApps[a].ops = new Array;
        var ops = apps[a].getElementsByTagName("op");
        for (o=0;o<ops.length;o++) {
            thisOp = ops[o];
            // note we will be making a sparsely filled array
            opid = thisOp.getAttribute("id");
            dbApps[a].ops[opid] = new Object;
            appop = dbApps[a].ops[opid];
            appop.id = opid;
            appop.job = thisOp.getAttribute("job");
            appop.arr = thisOp.getAttribute("arr");
            appop.wkstn = thisOp.getElementsByTagName("wkstn")[0].text;
            appop.desc = thisOp.getElementsByTagName("desc")[0].text;
            appop.pred = new Array;
            appop.succ = new Array;
            refs = thisOp.getElementsByTagName("pred");
            for (r=0;r<refs.length;r++) {
                appop.pred[r] = new Object;
                appop.pred[r].appid = refs[r].getElementsByTagName("aid")[0].text;
                appop.pred[r].opid = refs[r].getElementsByTagName("opid")[0].text;
            }
            refs = thisOp.getElementsByTagName("succ");
            for (r=0;r<refs.length;r++) {
                appop.succ[r] = new Object;
                appop.succ[r].appid = refs[r].getElementsByTagName("aid")[0].text;
                appop.succ[r].opid = refs[r].getElementsByTagName("opid")[0].text;
            }
        }
    }
    var t2 = new Date();
    var time consumed = t2.getTime() - t1.getTime();
    return {
        dbApps: dbApps,
        time: time consumed
    };
}