Minimal Version Selection

Go & Versioning, Part 4

Russ Cox

February 21, 2018

research.swtch.com/vgo-mvs

A versioned Go command must decide which module versions to use in each build. I call this list of modules and versions for use in a given build the *build list*. For stable development, today's build list must also be tomorrow's build list. But then developers must also be allowed to change the build list: to upgrade all modules, to upgrade one module, or to downgrade one module.

The *version selection* problem therefore is to define the meaning of, and to give algorithms implementing, these four operations on build lists:

1. Construct the current build list.
2. Upgrade all modules to their latest versions.
3. Upgrade one module to a specific newer version.
4. Downgrade one module to a specific older version.

The last two operations specify one module to upgrade or downgrade, but doing so may require upgrading, downgrading, adding, or removing other modules, ideally as few as possible, to satisfy dependencies.

This post presents *minimal version selection*, a new, simple approach to the version selection problem. Minimal version selection is easy to understand and predict, which should make it easy to work with. It also produces *high-fidelity builds*, in which the dependencies a user builds are as close as possible to the ones the author developed against. It is also efficient to implement, using nothing more complex than recursive graph traversals, so that a complete minimal version selection implementation in Go is only a few hundred lines of code.

Minimal version selection assumes that each module declares its own dependency requirements: a list of minimum versions of other modules. Modules are assumed to follow the import compatibility rule—packages in any newer version should work as well as older ones—so a dependency requirement gives only a minimum version, never a maximum version or a list of incompatible later versions.

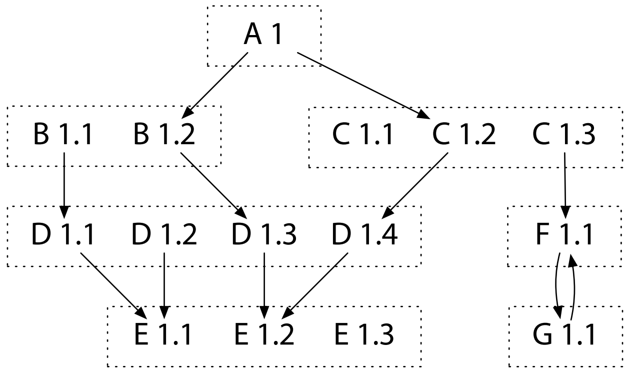
Then the definitions of the four operations are:

1. To construct the build list for a given target: start the list with the target itself, and then append each requirement's own build list. If a module appears in the list multiple times, keep only the newest version.
2. To upgrade all modules to their latest versions: construct the build list, but read each requirement as if it requested the latest module version.
3. To upgrade one module to a specific newer version: construct the non-upgraded build list and then append the new module's build list. If a module appears in the list multiple times, keep only the newest version.
4. To downgrade one module to a specific older version: rewind the required version of each top-level requirement until that requirement's build list no longer refers to newer versions of the downgraded module.

These operations are simple, efficient, and easy to implement.

Example

Before we examine minimal version selection in more detail, let's look at why a new approach is necessary. We'll use the following set of modules as a running example throughout the post:



The diagram shows the module requirement graph for seven modules (dotted boxes) with one or more versions. Following semantic versioning, all versions of a given module share a major version number. We are developing module A 1, and we will run commands to update its dependency requirements. The diagram shows both A 1's current requirements and the requirements declared by various versions of released modules B 1 through F 1.

Because the major version is part of the module's identifier, we must know that we are working on A 1 as opposed to A 2, but otherwise the exact version of A is unspecified—our work is unreleased. Similarly, different major versions are just different modules: for the purposes of these algorithms, B 1 is no more related to B 2 than to C 1. We could replace B 1 through F 1 in the diagram with A 2 through A 7 at a significant loss in clarity but without any change in how the algorithms handle the example. Because all the modules in the example do have major version 1, from now on we will omit the major version when possible, shortening A 1 to A. Our current development copy of A requires B 1.2 and C 1.2. B 1.2 in turn requires D 1.3. An earlier version, B 1.1, required D 1.1. And so on. Note that F 1.1 requires G 1.1, but G 1.1 also requires F 1.1. Declaring this kind of cycle can be important when singleton functionality moves from one module to another. Our algorithms must not assume the module requirement graph is acyclic.

Low-Fidelity Builds

Go's current version selection algorithm is simplistic, providing two different version selection algorithms, neither of which is right.

The first algorithm is the default behavior of go `get`: if you have a local version, use that one, or else download and use the latest version. This mode can use versions that are too old: if you have B 1.1 installed and run go `get` to download A, go `get` would not update to B 1.2, causing a failed or buggy build.

The second algorithm is the behavior of go `get -u`: download and use the latest version of everything. This mode fails by using versions that are too new: if you run go `get -u` to download A, it will correctly update to B 1.2, but it will also update to C 1.3 and E 1.3, which aren't what A asks for, may not have been tested, and may not work.

I call both these outcomes low-fidelity builds: viewed as attempts to reproduce the build that A's author used, these builds differ for no good reason. After we've seen the details of the minimal version selection algorithms, we'll look at why they produce high-fidelity builds instead.

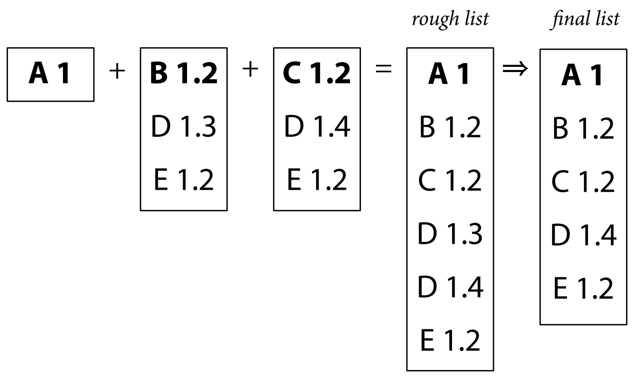
Algorithms

Now let's look at the algorithms in more detail.

Algorithm 1: Construct Build List

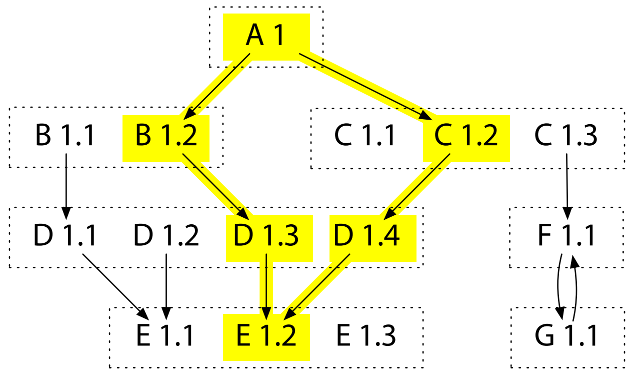
There are two useful (and equivalent) ways to define build list construction: as a recursive process and as a graph traversal.

The recursive definition of build list construction is as follows. Construct the rough build list for M by starting an empty list, adding M, and then appending the build list for each of M's requirements. Simplify the rough build list to produce the final build list, by keeping only the newest version of any listed module.



The recursive construction of build lists is useful mainly as a mental model. A literal implementation of that definition would be too inefficient, potentially requiring time exponential in the size of an acyclic module requirement graph and running forever on a cyclic graph.

An equivalent, more efficient construction is based on graph reachability. The rough build list for M is also just the list of all modules reachable in the requirement graph starting at M and following arrows. This can be computed by a trivial recursive traversal of the graph, taking care not to visit a node that has already been visited. For example, A's rough build list is the highlighted module versions found by starting at A and following the highlighted arrows:



(The simplification from rough build list to final build list remains the same.)

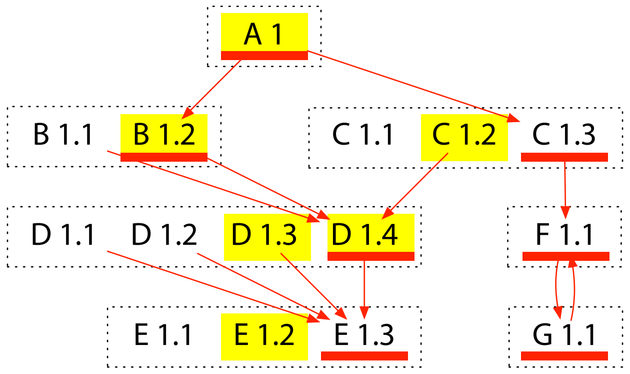
Note that this algorithm only visits each module in the rough build list once, and only those modules, so the execution time is proportional to the rough build list size $|B|$ plus the number of arrows that must be traversed (at most $|B|^2$). The algorithm completely ignores versions left off the rough build list: for example, it loads information about D 1.3, D 1.4, and E 1.2, but it does not load information about D 1.2, E 1.1 or E 1.3. In a dependency management setting, where loading information about each module version may mean a separate network round trip, avoiding unnecessary module versions is an important optimization.

Algorithm 2. Upgrade All Modules

Upgrading all modules is perhaps the most common modification made to build lists. It is what `go get -u` does today.

We compute an upgraded build list by upgrading the module requirement graph and then applying the previous algorithm. An upgraded module requirement graph is one in which every arrow pointing at any version of a module has been replaced by one pointing at the latest version of that module. (It is then also possible to discard all older versions from the graph, but the build list construction won't look at them anyway, so there's no need to clean up the graph.)

For example, here is the upgraded module requirement graph, with the original build list still marked in yellow and the upgraded build list now marked in red:



Although this tells us the upgraded build list, it does not yet tell us how to cause future builds to use that build list instead of the old build list (still marked in yellow). To upgrade the graph we changed the requirements for all modules, but an upgrade during development of module A must somehow be recorded only in A's requirement list (in A's `go.mod` file) in a way that causes Algorithm 1 to produce the build list we want, to pick the red modules instead of the yellow ones. To decide what to add to A's requirement list to cause that effect, we introduce a helper, Algorithm R.

Algorithm R. Compute a Minimal Requirement List

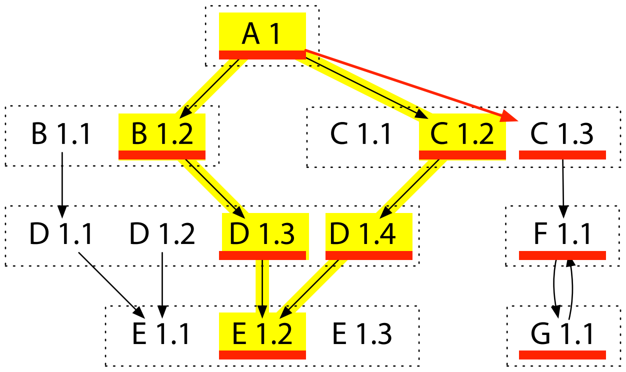
Given a build list compatible with the module requirement graph below the target, we want to compute a requirement list for the target that will induce that build list. It is always sufficient to list every module in the build list other than the target itself. For example, the upgrade we considered above could add C 1.3 (replacing C 1.2), D 1.4, E 1.3, F 1.1, and G 1.1 to A's requirement list. But in general not all of these additions are necessary, and we want to list as few additional modules as possible. For example, F 1.1 implies G 1.1 (and vice versa), so we need not list both. At first glance it seems natural to start by adding the module versions marked in red but not yellow (on the new list but missing from the old list). That heuristic would incorrectly drop D 1.4, which is implied by the old requirement C 1.2 but not by the new requirement C 1.3.

Instead, it is correct to visit the modules in reverse postorder—that is, to visit a module only after considering all modules that point into it—and only keep a module if it is not implied by modules already visited. For an acyclic graph, the result is a unique, minimal set of additions. For a cyclic graph, the reverse-postorder traversal must break cycles, and then the set of additions is unique and minimal for the modules not involved in cycles. As long as the result is correct and stable, we'll accept non-minimal answers in the case of cycles. In this example, the upgrade needs to add C 1.3 (replacing C 1.2), D 1.4, and E 1.3. It can drop F 1.1 (implied by C 1.3) and G 1.1 (also implied by C 1.3).

Algorithm 3. Upgrade One Module

Instead of upgrading all modules, cautious developers typically want to upgrade only one module, with as few other changes to the build list as possible. For example, we may want to upgrade to C 1.3, and we do not want that operation to make unnecessary changes like upgrading to E 1.3. Like in Algorithm 2, we can upgrade one module by upgrading the requirement graph, constructing a build list from it (Algorithm 1), and then reducing that list back to a set of requirements for the top-level module (Algorithm R). To upgrade the requirement graph, we add one new arrow from the top-level module to the upgraded module version.

For example, if we want to change A's build to upgrade to C 1.3, here is the upgraded requirement graph:



Like before, the new build list's modules are marked in red, and the old build list's are in yellow.

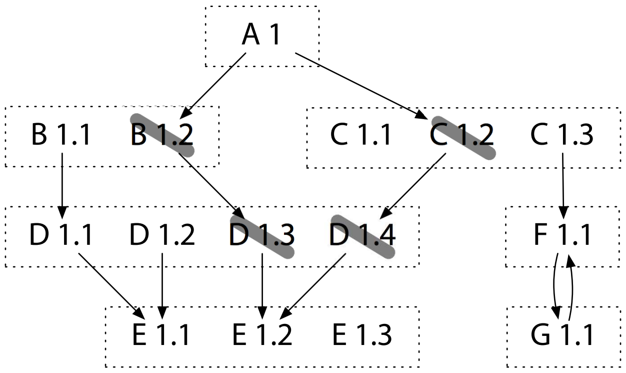
The upgrade's effect on the build list is the unique minimal way to make the upgrade, adding the new module version and any implied requirements but nothing else. Note that when constructing the upgraded graph, we must only add new arrows, not replace or remove old ones. For example, if the new arrow from A to C 1.3 replaced the old arrow from A to C 1.2, the upgraded build list would omit D 1.4. That is, the upgrade of C would downgrade D, an unexpected, unwanted, and non-minimal change. Once we've computed the build list for the upgrade, we can run Algorithm R (above) to decide how to update the requirements list. In this case we'd end up replacing C 1.2 with C 1.3 but then also adding a new requirement on D 1.4, to avoid the accidental downgrade of D. Note that this selective upgrade only updates other modules to C's minimum requirements: the upgrade of C does not simply fetch the latest of each of C's dependencies.

Algorithm 4. Downgrade One Module

We may also discover, perhaps after upgrading all modules, that the latest module version is buggy and must be avoided. In that situation, we need to be able to downgrade to an earlier version of the module. Downgrading one module may require downgrading other modules, but we want to downgrade as few other modules as possible. Like upgrades, downgrades must make their changes to the build list by modifying a target's requirements list. Unlike upgrades, downgrades must work by removing requirements, not adding them. This observation leads to a very simple downgrade algorithm that considers each of the target's requirements individually. If a requirement is incompatible with the proposed downgrade—that is, if the requirement's build list includes a now-disallowed module version—then try successively older versions until finding one that is compatible with the downgrade.

For example, starting with the original build graph, suppose we discover that there is a problem with D 1.4, actually introduced in D 1.3, and so we decide to downgrade to D 1.2. Our target module A depends on B 1.2 and C 1.2. To downgrade from D 1.4 to D 1.2, we must find earlier versions of B and C that do not require (directly or indirectly) versions of D later than D 1.2.

Although we can consider each requirement separately, it is more efficient to consider the module requirement graph as a whole. In our example, the downgrade rule amounts to crossing out the unavailable versions of D and then following arrows backwards from unavailable modules to find and cross out other unavailable modules. At the end, the latest versions of A's requirements that remain can be recorded as the new requirements.



In this case, downgrading to D 1.2 implies downgrading to B 1.1 and C 1.1. To avoid an unnecessary downgrade to E 1.1, we must also add a new requirement on E 1.2. We can apply Algorithm R to find the minimal set of new requirements to write to `go.mod`.

Note that if we'd first upgraded to C 1.3, then the downgrade to D 1.2 would have continued to use C 1.3, which doesn't use any version of D at all. But downgrades are constrained to only downgrade packages, not also upgrade them; if an upgrade before downgrade is needed, the user must ask for it explicitly.

Theory

Minimal version selection is *very* simple. It achieves simplicity by eliminating all flexibility about what the answer must be: the build list is exactly the versions specified in the requirements. A real system needs more flexibility, for example the ability to exclude certain module versions or replace others. Before we add those, it is worth examining the theoretical basis for the current system's simplicity, so we understand which kinds of extensions preserve that simplicity and which do not.

If you are familiar with the way most other systems approach version selection, or if you remember my Version SAT post from a year ago, probably the most striking feature of Minimal version selection is that it does not solve general Boolean satisfiability, or SAT. As I explained in my earlier post, it takes very little for a version search to fall into solving SAT; version searches in these systems are inherently intricate, complex problems for which we know no general efficient solutions. If we want to avoid this fate, we need to know where the boundaries are, where not to step as we explore the design space. Conveniently, Schaefer's Dichotomy Theorem describes those boundaries precisely. It identifies six restricted classes of Boolean formulas for which satisfiability can be decided in polynomial time and then proves that for any class of formulas beyond those, satisfiability is NP-complete. To avoid NP-completeness, we need to limit the version selection problem to stay within one of Schaefer's restricted classes.

It turns out that minimal version selection lies in the intersection of three of the six tractable SAT subproblems: 2-SAT, Horn-SAT, and Dual-Horn-SAT. The formula corresponding to a build in minimal version selection is the AND of a set of clauses, each of which is either a single positive literal (this version must be installed, such as during an upgrade), a single negative literal (this version is not available, such as during a downgrade), or the OR of one negative and one positive literal (an implication: if this version is installed, this other version must also be installed). The formula is a 2-CNF formula, because each clause has at most two variables. The formula is also a Horn formula, because each clause has at most one positive literal. The formula is also a dual-Horn formula, because each clause has at most one negative literal. That is, every satisfiability problem posed by minimal version selection can be solved by your choice of three different efficient algorithms. It is even simpler and more efficient to specialize further, as we did above, taking advantage of the very limited structure of these problems.

Although 2-SAT is the most well-known example of a SAT subproblem with an efficient solution, the fact that these problems are both Horn and dual-Horn formulas is more interesting. Every Horn formula has a unique satisfying assignment with the fewest variables set to true. This proves that there is a unique minimal answer for constructing a build list, as well for each upgrade. The unique minimal upgrade does not use a newer version of a given module unless absolutely necessary. Conversely, every dual-Horn formula also has a unique satisfying assignment with the fewest variables set to *false*. This proves that there is a unique minimal answer for each downgrade. The unique minimal downgrade does not use an older version of a given module unless absolutely necessary. If we want to extend minimal version selection, for example with the ability to exclude certain modules, we can only keep the uniqueness and mimimality properties by continuing to use constraints expressible as both Horn and dual-Horn formulas.

(Digression: The problem minimal version selection solves is NL-complete: it's in NL because it's a subset of 2-SAT, and it's NL-hard because st-connectivity can be trivially transformed into a minimal version selection build list construction problem. It's delightful that we've replaced an NP-complete problem with an NL-complete problem, but there's little practical value to knowing that: being in NL only guarantees a polynomial-time solution, and we already have a linear-time one.)

Excluding Modules

Minimal version selection always selects the minimal (oldest) module version that satisfies the overall requirements of a build. If that version is buggy in some way, an upgrade or downgrade operation can modify the top-level target's requirements list to force selection of a different version.

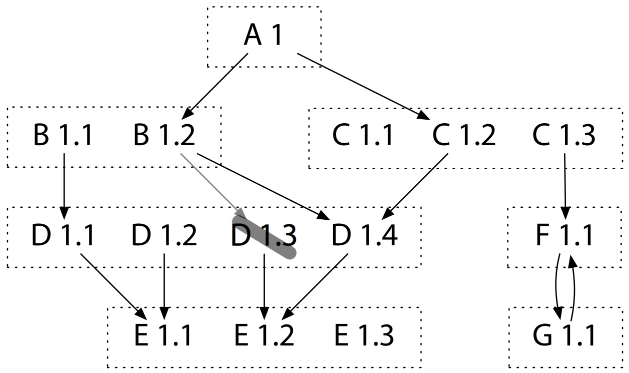
It can also be useful to record explicitly that the version is buggy, to avoid reintroducing it in any future upgrade or downgrade operations. But we want to do that in a way that keeps the uniqueness and minimality properties of the previous section, so we must use constraints that are both Horn and dual-Horn formulas. That means build constraints can only be unconditional positive assertions (X: X must be installed), unconditional negative assertions ($\neg$ Y: Y must not be installed), and positive implications (X $\rightarrow$ Z, equivalently $\neg$ X $\vee$ Z: if X is installed, then Z must be installed). Negative implications (X $\rightarrow$ $\neg$ Y, equivalently $\neg$ X $\vee$ $\neg$ Y: if X is installed, then Y must *not* be installed) cannot be added as constraints without breaking the form. Module exclusions must therefore be unconditional: they must be decided independent of selections made during build list construction.

What we *can* do is allow a module to declare its own *local* list of excluded module versions. By local, I mean that the list is consulted only for builds within that module; a larger build using the module only as a dependency would ignore the exclusion list. In our example, if A's build consulted D 1.3's list, then the exact set of exclusions would depend on whether the build selected, say, D 1.3 or D 1.4, making the exclusions conditional and leading to an NP-complete search problem. Only the top-level module is guaranteed to be in the build, so only the top-level module's exclusion list is used. Note that it would be fine to consult exclusion lists from other sources, such as a global exclusion list loaded over the network, as long as the decision to use the list is made before the build begins and the list content does not depend on which modules are selected during the build.

Despite all the focus on making exclusions unconditional, it might seem like we already have conditional exclusions: C 1.2 requires D 1.4 and so implicitly excludes D 1.3. But our algorithms do not treat this as an exclusion. When Algorithm 1 runs, it adds both D 1.3 (for B) and D 1.4 (for C) to the rough build list, along with their minimum requirements. The final simplification pass removes D 1.3 only because D 1.4 is present. The difference here between declaring an incompatibility and declaring a minimum requirement is critical. Declaring that C 1.2 must not be built with D 1.3 only describes how to fail. Declaring that C 1.2 must be built with D 1.4 instead describes how to succeed.

Exclusions then must be unconditional. Knowing that fact is important, but it does not tell us exactly how to implement exclusions. A simple answer is to add exclusions as the build constraints, with clauses like "D 1.3 must not be installed." Unfortunately, adding that clause alone would make modules that require D 1.3, like B 1.2, uninstallable. We need to express somehow that B 1.2 can choose D 1.4. The simple way to do that is to revise the build constraint, changing "B 1.2 → D 1.3" to "B 1.2 → D 1.3 ∨ D 1.4" and in general allowing all future versions of D. But that clause (equivalently, ¬ B 1.2 ∨ D 1.3 ∨ D 1.4) has two positive literals, making the overall build formula not a Horn formula anymore. It is still a dual-Horn formula, so we can still define a linear-time build list construction, but that construction—and therefore the question of how to perform an upgrade—would no longer be guaranteed to have a unique, minimal answer.

Instead of implementing exclusions as new build constraints, we can implement them by changing existing ones. That is, we can modify the requirements graph, just as we did for upgrades and downgrades. If a specific module is excluded, then we can remove it from the module requirement graph but also change any existing requirements on that module to require the next newer version instead. For example, if we excluded D 1.3, then we'd also update B 1.2 to require D 1.4:



If the latest version of a module is removed, then any modules requiring that version also need to be removed, as in the downgrade algorithm. For example, if G 1.1 were removed, then C 1.3 would need to be removed as well.

Once the exclusions have been applied to the module requirement graph, the algorithms proceed as before.

Replacing Modules

During development of A, suppose we find a bug in D 1.4, and we want to test a potential fix. We need some way to replace D 1.4 in our build with an unreleased copy U. We can allow a module to declare this as a replacement: "proceed as if D 1.4's source code and requirements have been replaced by U's."

Like exclusions, replacements can be implemented by modifying the module requirement graph in a preprocessing step, not by adding complexity to the algorithms that process the graph. Also like exclusions, the replacement list is local to one module. The build of A consults the replacement list from A but not from B 1.2, C 1.2, or any of the other modules in the build. This avoids making replacements conditional, which would be difficult to implement, and it also avoids the possibility of conflicting replacements: what if B 1.2 and C 1.2 specify different replacements for E 1.2? More generally, keeping exclusions and replacements local to one module limits the control that module exerts on other builds.

Who Controls Your Build?

The dependencies of a top-level module must be given some control over the top-level build. B 1.2 needs to be able to make sure it is built with D 1.3 or later, not with D 1.2. Otherwise we end up with the current go `get`'s stale dependency failure mode.

At the same time, for builds to remain predictable and understandable, we cannot give dependencies arbitrary, fine-grained control over the top-level build. That leads to conflicts and surprises. For example, suppose B declares that it requires an even version of D, while C declares that it requires a prime version of D. D is frequently updated and is up to D 1.99. Using B or C in isolation, it's always possible to use a relatively recent version of D (D 1.98 or D 1.97, respectively). But when A uses both B and C, the build silently selects the much older (and buggier) D 1.2 instead. That's an extreme example, but it raises the question: why should the authors of B and C be given such extreme control over A's build? As I write this post, there is an open bug report that the Kubernetes Go client declares a requirement on a specific, two-year-old version of `gopkg.in/yaml.v2`. When a developer tried to use a new feature of that YAML library in a program that already used the Kubernetes Go client, even after attempting to upgrade to the latest possible version, code using the new feature failed to compile, because "latest" had been constrained by the Kubernetes requirement. In this case, the use of a two-year-old YAML library version may be entirely reasonable within the context of the Kubernetes code base, and clearly the Kubernetes authors should have complete control over their own builds, but that level of control does not make sense to extend to other developers' builds.

In the design of module requirements, exclusions, and replacements, I've tried to balance the competing concerns of allowing dependencies enough control to ensure a succesful build without allowing them so much control that they harm the build. Minimum requirements combine without conflict, so it is feasible (even easy) to gather them from all dependencies. But exclusions and replacements can and do conflict, so we allow them to be specified only by the top-level module.

A module author is therefore in complete control of that module's build when it is the main program being built, but not in complete control of other users' builds that depend on the module. I believe this distinction will make minimal version selection scale to much larger, more distributed code bases than existing systems.

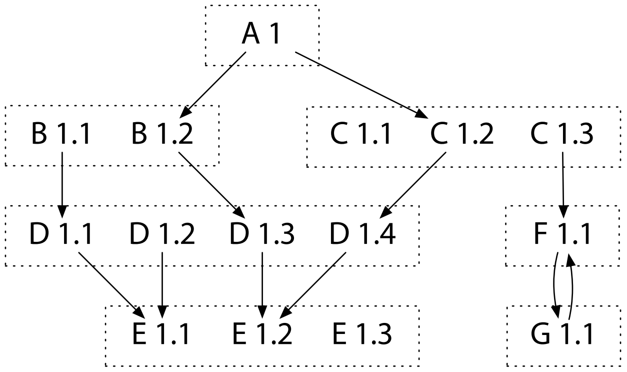
High-Fidelity Builds

Let's return now to the question of high-fidelity builds.

At the start of the post we saw that, using `go get` to build A, it was possible to use dependencies different than the ones A's author had used, without a good reason. I called this as a low-fidelity build, because it is a poor reproduction of the original build of A. Using minimal version selection, builds are instead high-fidelity. The module requirements, which are included with the module's source code, uniquely determine how to build it directly. The user's build of A will match the author's build exactly: a reproducible build. But high-fidelity means more.

Having a reproducible build is generally understood to be a binary property, for a whole-program build: a user's build is exactly the same the author's, or it isn't. What about when building a library module as part of a larger program? It would be helpful for a user's build of a library to match the author's whenever possible. Then the user runs the same code (including dependencies) that the author developed and tested with. In a larger project, of course, it may be impossible for a user's build of a library to match the author's build exactly. Another part of that build may force the use of a newer dependency, making the user's build of the library deviate from the author's build. Let's refer to a build as high-fidelity when it deviates from the author's own build only to satisfy a requirement elsewhere in the build.

Consider again our original example:



In this example, the build of A combines B 1.2 and D 1.4, even though B's author was using D 1.3. That change is necessary because A also uses C 1.2, which requires D 1.4. The build of A is still a high-fidelity build of B 1.2: it deviates by using D 1.4, but only because it must. In contrast, if the build used E 1.3, as `go get -u`, Dep, and Cargo typically do, that build would be low-fidelity: it deviates unnecessarily.

Minimal version selection provides high-fidelity builds by using the oldest version available that meets the requirements. The release of a new version has no effect on the build. In contrast, most other systems, including Cargo and Dep, use the newest version available that meets requirements listed in a "manifest file." The release of a new version changes their build decisions. To get reproducible builds, these systems add a second mechanism, the "lock file," which lists the specific versions a build should use. The lock file ensures reproducible builds for whole programs, but it is ignored for library modules; the Cargo FAQ explains that this is "precisely because a library should **not** be deterministically recompiled for all users of the library." It's true that a perfect reproduction is not always possible, but by giving up entirely, the Cargo approach admits unnecessary deviation from the library author's builds. That is, it delivers low-fidelity builds. In our example, when A first adds B 1.2 or C 1.2 to its build, Cargo will see that they require E 1.2 or later and will choose E 1.3. Until directed otherwise, however, it seems better to continue to build with E 1.2, as the authors of B and C did. Using the oldest allowed version also eliminates the redundancy of having two different files (manifest and lock) that both specify which modules versions to use.

Automatically using newer versions also makes it easy for minimum requirements to be wrong. Suppose we start working on A using B 1.1, the latest version at the time, and we record that A requires only B 1.1. But then B 1.2 comes out and we start using it in our own builds and lock file, without updating the manifest. At this point there is no longer any development or testing of A with B 1.1. We may start using features or depending on bug fixes from B 1.2, but now A incorrectly lists its minimum requirement as B 1.1. If users always also choose newer versions than the minimum requirement, then there is not much harm done: they'll use B 1.2 as well. But when the system does try to use the declared minimum, it will break. For example, when a user attempts a limited update of A, the system cannot see that updating to B 1.2 is also required. More generally, whenever the minimum versions (in the manifest) and the built versions (in the lock file) differ, why should we believe that building with the minimum versions will produce a working library? To try to detect this problem, Cargo developers have proposed that `cargo publish` try a build with the minimum versions of all dependencies before publishing. That will detect when A starts using a new feature in B 1.2—building with B 1.1 will fail—but it will not detect when A starts depending on a new bug fix.

The fundamental problem is that preferring the newest allowed version of a module during version selection produces a low-fidelity build. Lock files are a partial solution, targeting whole-program builds; additional build checks like in `cargo publish` are also a partial solution. A more complete solution is to use the version of the module the author did. That makes a user's build as close as possible to the author's build: a high-fidelity build.

Upgrade Speed

Given that minimal version selection takes the minimum allowed version of each dependency, it's easy to think that this would lead to use of very old copies of packages, which in turn might lead to unnecessary bugs or security problems. In practice, however, I think the opposite will happen, because the minimum allowed version is the *maximum* of all the constraints, so the one lever of control made available to all modules in a build is the ability to force the use of a newer version of a dependency than would otherwise be used. I expect that users of minimal version selection will end up with programs that are almost as up-to-date as their friends using more aggressive systems like Cargo.

For example, suppose you are writing a program that depends on a handful of other modules, all of which depend on some very common module, like `gopkg.in/yaml.v2`. Your program's build will use the *newest* YAML version among the ones requested by your module and that handful of dependencies. Even just one conscientious dependency can force your build to update many other dependencies. This is the opposite of the Kubernetes Go client problem I mentioned earlier.

If anything, minimal version selection would instead suffer the opposite problem, that this "max of the minimums" answer serves as a ratchet that forces dependencies forward too quickly. But I think in practice dependencies will move forward at just the right speed, which ends up being just the right amount slower than Cargo and friends.

Upgrade Timing

A key feature of minimal version selection is that upgrade do not happen until a developer asks for them to happen. You don't get an untested version of a module unless you asked for that module to be upgraded.

For example, in Cargo, if package B depends on package C 2.9 and you add B to your build, you don't get C 2.9. You get the latest allowed version at that moment, maybe C 2.15. Maybe C 2.15 was released just a few minutes ago and the author hasn't yet been told about an important bug. That's too bad for you and your build. On the other hand, in minimal version selection, module B's `go.mod` file will list the exact version of C that B's author developed and tested with. You'll get that version. Or maybe some other module in your program developed and tested with a newer version of C. Then you'll get that version. But you will never get a version of C that some module in the program did not explicitly request in its `go.mod` file. This should mean you only ever get a version of C that worked for someone else, not the very latest version that maybe hasn't worked for anyone.

To be clear, my purpose here is not to pick on Cargo, which I think is a very well-designed system. I'm using Cargo here as an example of a model that many developers are familiar with, to try to convey what would be different in minimal version selection.

Minimality

I call this system minimal version selection because the system as a whole appears to be minimal: I don't see how to remove anything more without breaking it. Some people will undoubtedly say that too much has been removed already, but so far it seems perfectly able to handle the real-world cases I've examined. We'll find out more by experimenting with the `vgo` prototype.

The key to minimal version selection is its preference for the minimum allowed version of a module. When I compared `go get -u`'s "upgrade everything to latest" approach to Cargo's "manifest and lock" approach in the context of a system that can rely on the import compatibility rule, I realized that both manifest and lock exist for the same purpose: to work around the "upgrade everything to latest" default behavior. The manifest describes which newer versions are unneeded, and the lock describes which newer versions are unwanted. Instead, why not change the default? Use the minimum version allowed, typically the exact version the author used, and leave timing of upgrades completely to user control. This approach leads to reproducible builds without lock files, and more generally to high-fidelity builds that deviate from the author's own build only when required.

More than anything else, I wanted to find a version selection algorithm that was understandable. Predictable. Boring. Where other systems instead seem to optimize for displays of raw flexibility and power, minimal version selection aims to be invisible. I hope it succeeds.