

600.325/425 — Declarative Methods

Assignment 4: Dynamic Programming

Spring 2009
Prof. Jason Eisner

Due date: Monday, May 4, 2 pm

(Note: Remaining late days may only be used through Friday, May 8)

The questions in this assignment concern Dyna. Questions 4–5 are closely related to questions you did in the Prolog assignment.

Policies and general submission instructions are the same as for the Prolog assignment.

1. Let's review dynamic programming using the Knapsack algorithm. This is a pencil-and-paper problem.

As explained on the class slides, you are considering N objects with non-negative integer weights w_1, w_2, \dots, w_N and corresponding values v_1, v_2, \dots, v_N . So the subset $\{3, 4, 8\}$ would weigh $w_3 + w_4 + w_8$ and would have value $v_3 + v_4 + v_8$. Your knapsack can hold at most W in weight. So you would like to find the maximum-value subset with weight $\leq W$.

For $0 \leq n \leq N$ and any weight w , let $C[n, w]$ denote the maximum-value subset of $\{1, 2, \dots, n\}$ that has weight $\leq w$. So you would ultimately like to find $C[N, W]$.

Let $c[n, w]$ be the value of $C[n, w]$. In other words, $C[n, w]$ is a maximum-value subset and $c[n, w]$ is its actual value. Clear?

- (a) Write a formula for $c[n, w]$ in terms of other values of c , such as $c[n-1, w-w_n]$. Try to do it on your own, but use your class notes if necessary (we did this in class). Make sure to get the base cases right—computing $c[n, w]$ using this formula should not recurse forever!

- (b) Write procedural pseudocode for a *backward-chaining* function that computes $c[n, w]$. This program should be very short and should look a lot like your formula in part 1a. Make sure to memoize the result, though.

(If instead of pseudocode, you would prefer to write real code in a concise procedural language like Python or C, that is okay too. You are not required to compile it, unless you'd like to test it.)

- (c) Write procedural pseudocode that uses *forward chaining* (no memoization) to compute $c[n, w]$ for all integers $0 \leq n \leq N$ and all integers $0 \leq w \leq W$. So when you are all done, you will have $c[N, W]$ (among other things!). Make sure that you order your computation so that when you use a value like $c[n-1, w-w_n]$, it will already have been computed—you should never use an uninitialized value.
 - (d) What is the worst-case runtime of computing $c[N, W]$ using your forward-chaining program? Justify your answer (which should be in big- O notation, in terms of N and W).
 - (e) What is the worst-case runtime of computing $c[N, W]$ using your backward-chaining program? Justify your answer (which should be in big- O notation, in terms of N and W).
 - (f) How long would each method take if all the integer weights w_1, \dots, w_N happened to be divisible by 10? Explain.
 - (g) Now for the most useful part. Write pseudocode to print out $C[n, w]$ (which is the actual subset of items you should take, rather than just the value of that subset). You may want to call some modified version of the forward-chaining or backward-chaining code that you wrote in question 1b or 1c (your choice).
 - (h) [425] Your answer to question 1f should have established that backward chaining will sometimes be faster than forward chaining, depending on the weights. Modify your forward chaining pseudocode from question 1b to speed it up, by ensuring that it *always* computes *only* entries of $c[n, w]$ that backward chaining would compute. *Hint:* Let `interesting[n, w]` be true or false according to whether $c[n, w]$ needs to be derived. Write pseudocode to determine `interesting[n, w]` and then determine the relevant values of $c[n, w]$.
2. To find out how to run the Dyna compiler and debugger, see the “CS undergrad machines” section of <http://dyna.org/JHU>.

Now work through the start of the tutorial at <http://dyna.org/Tutorial>: “hello world,” Dijkstra’s algorithm, and the debugger.

Also read http://dyna.org/Several_perspectives_on_Dyna.

There is nothing to hand in for this question.

Important request: Please send feedback about Dyna’s usability to the `cs325-staff` email address. If the compiler or the visual debugger does something you don’t expect, or gave a confusing error message, please forward it to us. We need the feedback and are happy to help quickly.

Disclaimer: Dyna is both a language and an implementation. The prototype implementation that you are using is for an older version of the language, and has not been

updated for a few years except for minor fixes. The version under development is cleaner, faster, and more powerful—as you saw in lecture—but is being rebuilt from the ground up and is unfortunately not ready for you yet. Email the professor if you would like to receive announcements of new versions in the future.

The visual debugger, Dynasty, is also being rebuilt from the ground up. New features are in the works, but hopefully it’s usable at this point. There are still a few stability issues (mainly owing to bugs in the underlying toolkits that we use). Try restarting Dynasty if it hangs on you.

We apologize for any rough edges you encounter in this assignment. If you encounter a cryptic error message, something poorly explained in the documentation, or especially a bug, please don’t hesitate to inform us ASAP by email.

3. First, an easy problem to warm you up. A number of presidents of the United States have been blood relatives of one another:

- George Bush, George W. Bush (father and son)
- John Adams, John Q. Adams (father and son)
- Theodore Roosevelt, Martin van Buren (third cousins twice removed)
- Theodore Roosevelt, Franklin Roosevelt (fifth cousins)

It is natural to ask questions like “Who was the most recent common ancestor of Theodore and Martin, and how recent was he or she?”

Write a short, well-commented Dyna program, `ancestor.dyna`, to find the most recent common ancestor of two people.

You should be able to run the result as

```
./ancestor presidents.par queryA.par
```

These files (like others in this assignment) are available from either <http://cs.jhu.edu/~jason/325/hw4> or `/usr/local/data/cs325/hw4`. The `presidents.par` file contains both Roosevelt family data and Adams family data—plenty for you to experiment with. It’s traditional to call this a family tree, but actually it is a “family DAG” (directed acyclic graph).

You are supposed to find the *most recent* common ancestor (not Adam or Eve). We define the “recency” of a common ancestor to be the *total* length of his or her shortest paths from the two descendants. For example, <http://www.gwu.edu/~erpapers/teachinger/q-and-a/q6.htm> shows that Franklin and his wife Eleanor had a common ancestor of recency 13: namely Nicholas, who was 6 generations above Franklin

and 7 generations above Eleanor.¹ Eleanor's last name was Roosevelt even *before* she married Franklin!

The most recent common ancestor of Nicholas and Nicholas was Nicholas, with recency 0. (For our purposes, he was his own ancestor.)

Turn in your commented code in a file called `ancestor.dyna`. In your `README`, give the results of `queryA`, `queryB`, `queryC`, and `queryD` when you compile with `--driver=backtrace`.² Explain how to interpret these results. You could also try `--driver=dynasty_bestonly`.

It is okay (but unnecessary) to write your own driver program or alter the `.par` files. If you do these things, also turn in your changed files and alert us in your `README`.

Note: In the current version of Dyna (v0.3), you will probably need to include the following to avoid a runtime type error when you read the `.par` file.³

```
:- structure(child(string,string)).
```

Hint: An earlier version of `queryA.par` read

```
person1("FRANKLIN DELANO ROOSEVELT 1882-1945") := 0.  
person2("ANNA ELEANOR ROOSEVELT 1884-1962") := 0.
```

You may want to try solving the problem that way first. A hint:

¹If both of Nicholas's sons had the same mother, then she was another common ancestor of recency 13.

²Remember from the Dijkstra's algorithm tutorial that for backtracing to work, you will need to add the following line to your Dyna program: `:- pragma(keep_best_antecedents(item))`.

³If you don't include this line, the compiler concludes that the arguments of `child` can be arbitrary terms. That's great, except that, alas, native types like strings do not yet count as terms. They will in the near future. (Java faced a similar situation until Java 1.5 introduced automatic upcast from `int` to `Integer`, known as "autoboxing").

This was not a problem in the tutorial because `path.dyna` used literal strings in the body of the program, in a way that allowed the compiler to guess the correct type declaration `:- structure(child(string,string))`. So the compiled `path` program was able to read `flights.par`.

As long as you stay away from ints and strings, you will *never* have any type problems in the current Dyna implementation. *All* current type errors arise from the the fact that ints and strings aren't yet first-class terms, just as ints and chars have never been first-class Objects in Java.

This will be fixed in the next version of Dyna. In the meantime, you can do your own "boxing" if you like by consistently using the term `i(3)` in place of the raw integer `3`. (Or you could call it `three` or `s(s(s(z)))` if you like.) Similarly, you could use the term `s("Roosevelt")` in place of the raw string `"Roosevelt"`.

Basically, as long as you work with terms, Dyna acts like a weakly typed language, as Prolog is. This is easier for casual programming because it means you'll never get type errors. The strong typing system will kick in only to the extent that you *choose* to make explicit type declarations. Such declarations inform the compiler that terms are *restricted* to have particular shapes. (This permits the compiler to generate more efficient code, and may help the compiler catch your errors by complaining if you use terms that don't have these shapes. Those are the main reasons why strong typing is popular in programming languages.)

```
path_from_1_up_to(X) + path_from_2_up_to(X)
```

But you will notice that your code for handling `person1` and `path_from_1_up_to` is basically identical to the code for handling `person2` and `path_from_2_up_to`. Duplicate code is inelegant and hard to maintain. The fix is to have a variable that ranges over 1,2. To do this, you'll need to change `path_from_1_up_to(Name)` to `path(1,Name)`, etc. This will allow you to eliminate your duplicate code, while changing `queryA.par` back to the version provided:

```
person(1,"FRANKLIN DELANO ROOSEVELT 1882-1945") := 0.  
person(2,"ANNA ELEANOR ROOSEVELT 1884-1962") := 0.
```

Note: If we had more time for this assignment, I would ask you write a Dyna program to compute “consanguinity”—i.e., the degree of blood relationship between two people in a family tree, such as Franklin and Eleanor (who were cousins) or their children James and Anna (who were siblings but also cousins). This is fairly easy in “ordinary” family trees, but gets trickier if there is inbreeding. Ask me if you’re interested.

4. In assignment 4, you wrote Prolog code to find the longest increasing subsequence of a given list. (Strictly increasing, i.e., no duplicates.)

Suppose the input is `[3,5,2,6,7,4,9,1,8,0]`. We pointed out in assignment 4 that it would be a bad idea to generate all subsequences, such as `[3,5,6,4,9]`, and then keep only the ones that were increasing. There would be 2^n subsequences to generate and test.

Instead, you did the `<` checks as you went along. You never even built `[3,5,6,4,9]`—because you wouldn’t have been willing to stick `6` at the front of `[4,9]` at an earlier step.

So your Prolog code generated only the *increasing* subsequences only, and then picked the longest.

- (a) Though better, why was that still inefficient? Give an example of a length-10 list where even generating all *increasing* subsequences would be slow.
- (b) Someone suggests the following “greedy” recursive solution: “To build the longest increasing subsequence of `[3,5,2,6,7,4,9,1,8,0]`, first build the longest increasing subsequence of `[5,2,6,7,4,9,1,8,0]`, then glue `3` on the front if you can.”

```
LIS(list):  
1.  if (list.empty())  
2.    return []  
3.  else
```

```

4.     subproblem = list.rest()
5.     (* find just the best solution to subproblem—not all solutions as in the Prolog version! *)
6.     subsolution = LIS(subproblem)
7.     if (subsolution.empty() or list.first() < subsolution.first())
8.         return cons(list.first(), subsolution)
9.     else
10.        return subsolution

```

Try this function *by hand* on the list [3,5,2,6,7,4,9,1,8,0]. What is the big-O runtime? What answer does the function get?

- (c) Now you'll correct the above solutions, using dynamic programming.

As preparation, remember the in-class problem of maximum-weight independent set in a tree. (For simplicity, let's restrict to just a simplified *binary*-tree version.) The algorithm had to solve 4 recursive subproblems rather than 2. It wasn't enough just to get the max-weight independent set of the left subtree and also of the right subtree! Rather, in each of the two subtrees, we had to get the max-weight independent set *and* the max-weight *unrooted* independent set. Knowing that a partial solution was unrooted allowed us to determine that we could legally combine it with other partial solutions in certain ways.

In other words, we decided to solve a slightly harder problem than the original. In effect, we wrote MIS(tree, must_be_rooted). Then we called both MIS(tree, false) and MIS(tree, true), at different times. So MIS had a slightly more general problem to solve, but was able to solve it *by relying on recursive copies of itself that could also solve more general problems!*

(This trick is known as “strengthening the inductive hypothesis.” You've done proofs by induction—basically a proof that calls itself recursively. Sometimes the only way to write one is to decide to prove a stronger theorem than you were assigned, because otherwise the recursive call—the “inductive hypothesis”—won't establish all the results that you need in order to solve the original problem.)

Now go back to the LIS problem. What do you have to know about an increasing subsequence of the subproblem [5,2,6,7,4,9,1,8,0] in order to know whether you are allowed to glue 3 onto the front?

- (d) Given your answer to 4c, improve the pseudocode from 4b so that it will get the *correct* answer. Like MIS, your LIS will have an extra argument that is used in those recursive calls. And like MIS, it will have to call itself more than once.
- (e) i. How should you call your new two-argument LIS function if you want the longest increasing subsequence of [3,5,2,6,7,4,9,1,8,0]?
 ii. The fact that your LIS is multiply recursive suggests that it might benefit from dynamic programming (i.e., reuse of subsolutions). Give an example

of a subproblem $\text{LIS}(x, y)$ that must be solved at least twice during the call you just proposed.

- iii. What specific technique would avoid the duplicate computation?
 - iv. What happens to your runtime if you *don't* avoid the duplicate computation? (For extra credit, prove your answer.)
- (f) Your LIS function worked by backward chaining, starting with the original list and calling simpler subproblems as needed.

Write a Dyna program that does essentially the same computation by forward chaining, starting with simpler lists and building the solution up from there.

Hints:

- Roughly speaking, the Dyna program will solve various $\text{LIS}(x, y)$ problems, starting with $x = []$ and working up to bigger lists x .
- To ensure that it doesn't run forever, your program should confine that computation to "interesting" lists that are tails of the original input. We saw how to do this in class:

```
interesting(Xs) :- input(Xs).
```

```
interesting(Xs) :- interesting([X|Xs]).
```

```
% replace :- by the accumulation operator used in rest of program
```

Then add "whenever interesting(...)" to some of your program's other rules, to constrain their use.

- *Hint:* The basic idea is to compute items of the form $\text{lis}(x)$, where the value of $\text{lis}(x)$ should be the *length* of the longest increasing subsequence of x . (Use `--driver=backtrace` to reconstruct an actual subsequence of that length.) But you will have to strengthen the inductive hypothesis. So you should also, or instead, state recursive rules for computing items of the form $\text{lis}(x, y)$, where y tells you something *about* the actual best subsequence.

y should not actually specify a full subsequence, since then you'd face an exponential proliferation of possible items. The idea is to keep the number of items pretty small, by having each item $\text{lis}(x, y)$ summarize a lot of possible subsequences of x and record only the length of the best one.

- The current Dyna compiler, alas, does not yet let you write $A < B$ in a program. We have written a little hack for you in `lessthan.dyna`. Read that file carefully. Your `lis.dyna` can include its material as follows:

```
#include "lessthan.dyna"
```

Note: If you get a compiler error about "inconsistent type declarations," **email us your code and we'll help.**⁴

⁴Dyna is a typed language, for reasons of safety and efficiency. But by default, the compiler will try to

Turn in your commented code as `lis.dyna`. Also turn in a parameter file, `lis.par`, that specifies the `[3,5,2,6,7,4,9,1,8,0]` problem.

- (g) In your `README`, explain how to decode your program's output (e.g., on `lis.par`) when it is compiled with `--driver=backtrace`. In other words, how could you figure out what the best subsequence is? You might also want to use `--driver=dynasty_bestonly`, with the option "Tools / Show Node Values."
(Note that in practice, you wouldn't decode the backtrace text. The C++ chart object provides programmer-friendly methods that let you extract the same information. So it's cleaner to write your own C++ driver program.)
- (h) What does "strengthening the inductive hypothesis" correspond to in Dyna?

5. [425] In assignment 4, you wrote Prolog/ECLiPSe code to generate balanced binary search trees.

Balanced trees might not always be what we want in practice, though. If some of the keys in the tree will be searched for much more often than the rest, then it will be more efficient in the long run to store those keys closer to the root, even if this means pushing other keys further down.

Suppose that we have a fixed set of keys that we want to store in a search tree, and we know (or can guess) exactly how many times per day we will want to search for each key. We want to construct a search tree so that the *total* number of nodes visited per day is minimized.

Because of the property that a subtree of an optimal search tree is itself an optimal search tree (for the keys in the subtree), constructing optimal search trees can be solved efficiently by dynamic programming.

See http://www.cs.auckland.ac.nz/software/AlgAnim/opt_bin.html for a full exposition of the problem, a traditional dynamic programming algorithm for solving it, and a very helpful Java animation of the algorithm.

- (a) Translate the above algorithm into a running Dyna program. Turn in your commented code as `optBST.dyna`. Your program should look like a very concise statement of the algorithm.

Sample inputs `abcde.par`, `words1.par`, `words2.par`, `words3.par`, and `words4.par` can be found in the usual directory. These problems are taken from real data

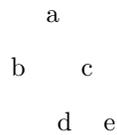
guess whatever type declarations you left out. If it can find equally strong arguments for different types, it will give up and ask you to include more declarations. (The current error message is misleading, since it sometimes means "insufficient" rather than "inconsistent.")

I ran into this when solving this question myself. We were already considering strengthening the type inference algorithm in such a way that it would have succeeded on my program. Meanwhile, adding the following line may be enough (it was for me): `:- structure(cons(int,list)).`

and are progressively larger. Since the algorithm has $O(n^3)$ runtime, the final input file might require half an hour to find the final answer.

Example: The following tree is *not* a solution to `abcde.par`, because it's not a legal search tree. When searching for a key in a subtree, we'd like to recurse to the left or right subtree according to whether the key is less or greater than the subtree's root. So we need a guarantee that the left (or right) subtree stores only keys that are less (or greater) than the root.

This means that in a legal search tree, the infix order of the nodes will be sorted, i.e., `[a,b,c,d,e]` rather than `[b,a,d,c,e]` as in the illegal search tree below:



However, this were a legal search tree, its cost for `abcde.par` would be

$$\begin{aligned}
 \text{cost} &= \sum_x (\text{time to find } x) \cdot (\text{how often we look for } x) \\
 &= \sum_x (\text{depth of } x) \cdot (\text{frequency of } x) \\
 &= 1 \cdot 50 + 2 \cdot 10 + 2 \cdot 20 + 3 \cdot 1 + 3 \cdot 3 \\
 &= 122
 \end{aligned}$$

- (b) In your `README`, explain how to decode your program's output (e.g., on `abcde.par`) when it is compiled with `--driver=backtrace`. In other words, how could you figure out what the optimal tree is?

For the small input files, you might want to try `--driver=dynasty.bestonly`, with the option "Tools / Show Node Values," or perhaps even `--driver=dynasty`, which shows all the ways of building each item, not only the best way.

- (c) The root of the optimal tree is the same word for all four of the `words*.par` files. What is this word and why is it so great? Is it the most frequent word? The most central word? If not, then what?
- (d) For this program, it is safe to add the declaration `:- converges(goal, first_pop)`. That says that one has found the true value of `goal` as soon as one has processed the first update to it. If one only wants to know `goal`, then it is not necessary to process further updates from the agenda until it empties out. How much does this speed things up?

(*Note:* One could get considerable further speedup by designing a better ordering for the agenda. The default is to process the lowest-value items first. A better class of solutions includes an A* heuristic. At the moment, this can be done only by writing a C++ function to define the priorities. Eventually,

though, the priority of an item `foo(123)` will be represented by another item `priority(foo(123))`, whose value can be defined and updated directly in Dyna like any other item.)

Hints:

- In the input files, a line like `key("d",3,4) := 15` encodes the statement that "d" is the 4th key in sorted order, and that you expect to search for it 15 times per day. You might expect just `key("d",4) := 15`. We include 3 in the item name merely as a trick to make it easy to combine this item `key("c",2,3)` that represents the previous key.
- Your job is to compute the value of items like `cost(2,5)`. This should represent the total cost of the best search subtree that spans all 3 consecutive keys `key("c",2,3)`, `key("d",3,4)`, and `key("e",4,5)`. (Note that $5 - 2 = 3$.) **The cost of a subtree is the total number of visits per day to nodes in that subtree.**

In general, `cost(I,J)` is the cost of the best subtree T spanning the range I to J . You may want to try writing a mathematical formula expressing this as a minimum over the different possible choices of breakpoint in T . Dyna notation is rather close to such formulas.

- Ultimately, you want to find the cost of the best search tree spanning all n keys. So write a rule that defines `goal` accordingly. It should use a clause like "whenever `numkeys(N)`."
- You'll want the declaration

```
:- structure(key(string,int,int)).
```

- Note that when you use a subtree as the left or right child of a larger tree, visiting any node in the subtree will also mean visiting the root node of the larger tree. How does that affect the cost of the larger tree? You'll need to strengthen your inductive hypothesis so that you have enough information to compute this new cost.

In this problem (unlike problem 4), strengthening the inductive hypothesis doesn't mean solving more-detailed subproblems, but rather returning more-detailed subsolutions. What does this correspond to in Dyna? This should give you a new way to answer question 4h.

- *Hint:* Avoid having too many items on the right-hand side of a rule, as in `a += b*c*d*e`. The way that you arrange and parenthesize them can have major consequences for speed, e.g., `a += b*(c*(d*e))` versus `a += (b*e)*(c*d)`.

Writing things out with temporary items will help you see which versions are more efficient.

For example, `a(X) += (b(W,X)*c(X,Y))*d(W)` is equivalent to

```
temp(W,X,Y) += b(W,X)*c(X,Y). % creates a cubic number of temp items
a(X) += temp(W,X,Y)*d(W).      % sums over W and Y
```

whereas `a(X) += (d(W)*b(W,X))*c(X,Y)` gets the same answer at only quadratic cost, being equivalent to

```
temp(X) += d(W)*b(W,X).      % sums over W
a(X) += temp(X)*c(X,Y).      % sums over Y
```

A summary of what to turn in for this assignment:

- The `dynac.log.gz` file that was created in the directory where you ran `dynac`. This will not be used when grading, but will help us improve Dyna by counting how often different error messages were encountered by novice users. (If you did different problems in different directories, please submit all your log files.)
- Your `README` file, including clear, well-thought-out answers to all of the questions. You may want to include general comments about Dyna.
- Turn in **well-commented** Dyna code for problems 2, 3, and 4. **Explain** how your code works! (You can give explanation in your `README` if you prefer.)
- **DO NOT** submit binaries, C++ driver programs,⁵ or any of the auxiliary files created by `dynac`. Just submit the stuff you wrote yourself.

Good luck!

p.s. One of the goals of the Dyna project is to create a language that is easy to use, even for people not familiar with logic programming or dynamic programming. Any feedback you can offer as to how to achieve that better in the new version will be greatly appreciated. We apologize again that you are using an old version.

⁵Unless, of course, you wrote your own—but using standard drivers like `--driver=backtrace` should be sufficient for this assignment.