Dynamic Programming

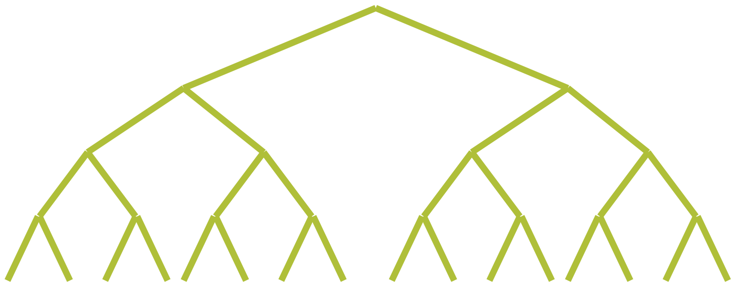
Logical re-use of computations

Divide-and-conquer

1. split problem into smaller problems
2. solve each smaller problem recursively
3. recombine the results

Divide-and-conquer

1. split problem into smaller problems
2. solve each smaller problem recursively
    1. split smaller problem into even smaller problems
    2. solve each even smaller problem recursively
        1. split smaller problem into eensy problems
        2. …
        3. …
    3. recombine the results
3. recombine the results



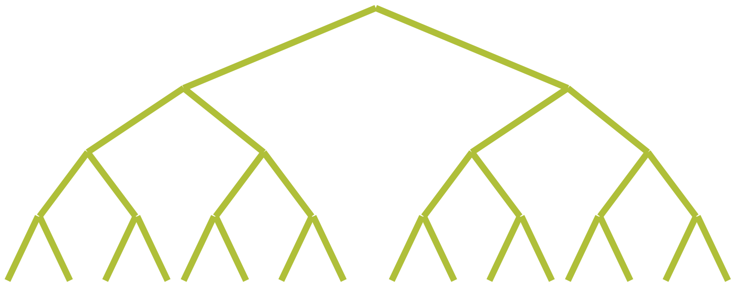
should remind you of backtracking

Dynamic programming

Exactly the same as divide-and-conquer …
but store the solutions to subproblems for possible reuse.

A good idea if many of the subproblems are the same as one another.

There might be $O(2^n)$ nodes in this tree, but only e.g. $O(n^3)$ *different* nodes.

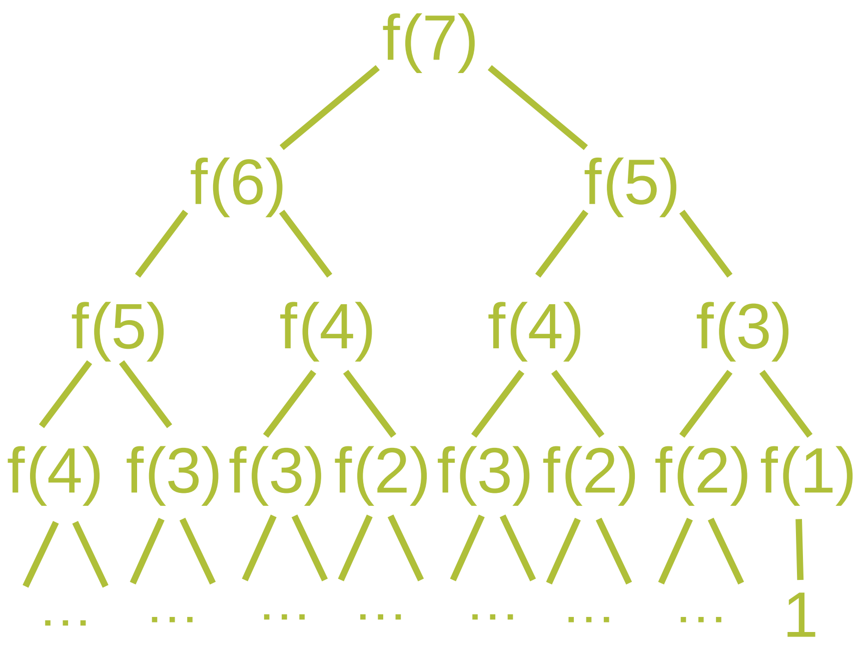


should remind you of backtracking

Fibonacci series

- 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- $f(0) = 0$.
- $f(1) = 1$.
- $f(N) = f(N-1) + f(N-2)$
  if $N \geq 2$.

```
int f(int n) {
    if n < 2
        return n
    else
        return f(n-1) + f(n-2)
}
```

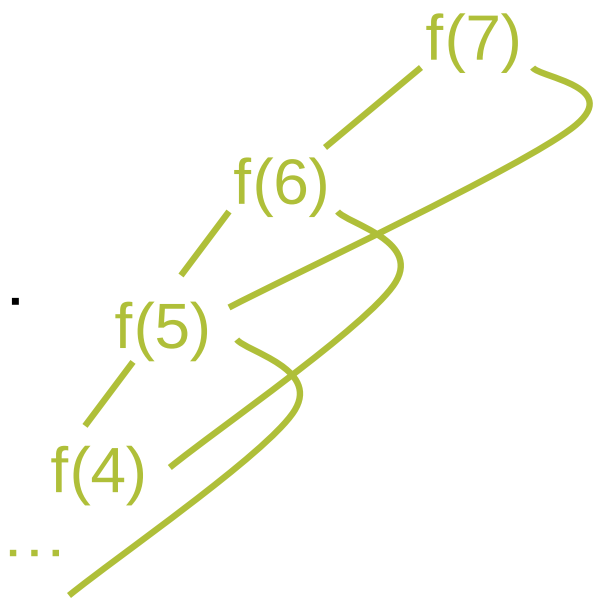


f(n) takes exponential time to compute.
Proof: f(n) takes more than twice as long as f(n-2), which therefore takes more than twice as long as f(n-4) …
Don't <u>you</u> do it faster?

Reuse earlier results!

("memoization" or "tabling")

- 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- f(0) = 0.
- f(1) = 1.
- f(N) = f(N-1) + f(N-2)
  if N ≥ 2.



```
int f(int n) {
    if n < 2
        return n
    else
        return $f_{memo}$(n-1) + $f_{memo}$(n-2)
}
```

does it matter which of these we call first?

```
int $f_{memo}$(int n) {
    if f[n] is undefined
        f[n] = f(n)
    return f[n]
}
```

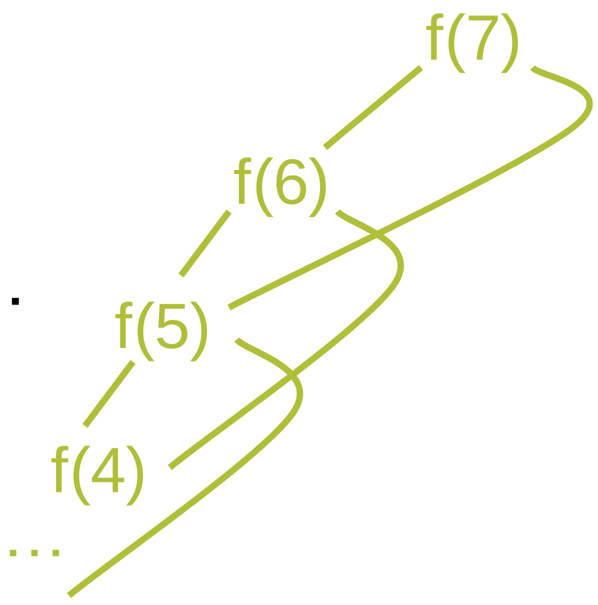
Backward chaining vs. forward chaining

- Recursion is sometimes called "backward chaining": start with the goal you want, f(7), choosing your subgoals f(6), f(5), … on an as-needed basis.
    - Reason backwards from goal to facts (start with goal and look for support for it)
- Another option is "forward chaining": compute each value as soon as you can, f(0), f(1), f(2), f(3) … in hopes that you'll reach the goal.
    - Reason forward from facts to goal (start with what you know and look for things you can prove)
- **Either way, you should table results that you'll need later**
- Mixing forward and backward is possible (future topic)

Reuse earlier results!

(forward-chained version)

- 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- f(0) = 0.
- f(1) = 1.
- f(N) = f(N-1) + f(N-2)
  if N ≥ 2.

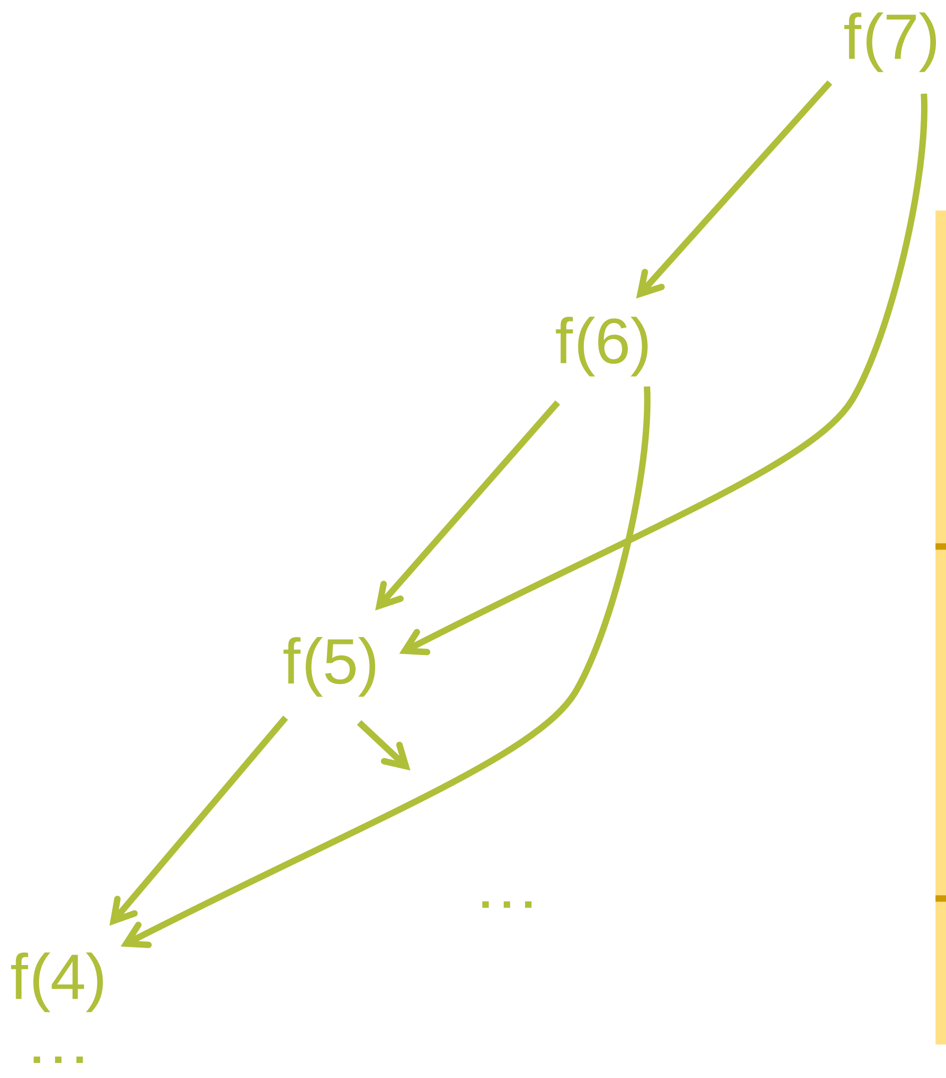


```
int f(int n) {
    f[0] = 0; f[1]=1
    for i=2 to n
        f[i] = f[i-1] + f[i-2]
    return f[n]
}
```

Which is more efficient, the forward-chained or the backward-chained version?

Can we make the forward-chained version even more efficient? (hint: save memory)

How to analyze runtime of backward chaining



Each node does some "local" computation to obtain its value from its children's values

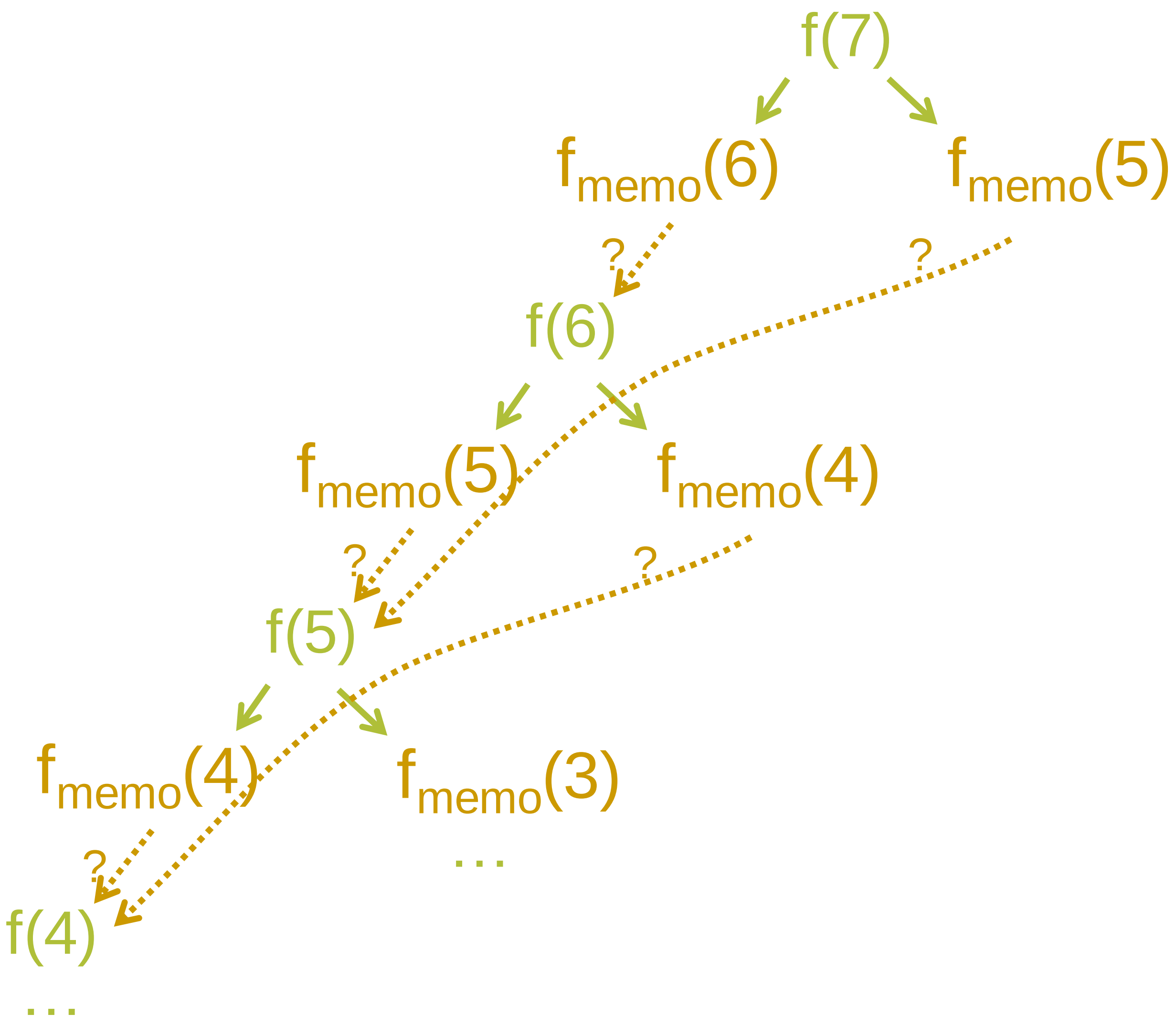
(often O(# of children))
(here, f(n) needs O(1) time itself)

Total runtime = total computation at all nodes that are visited

(often O(# of edges))
(here, f(n) needs O(n) total time)

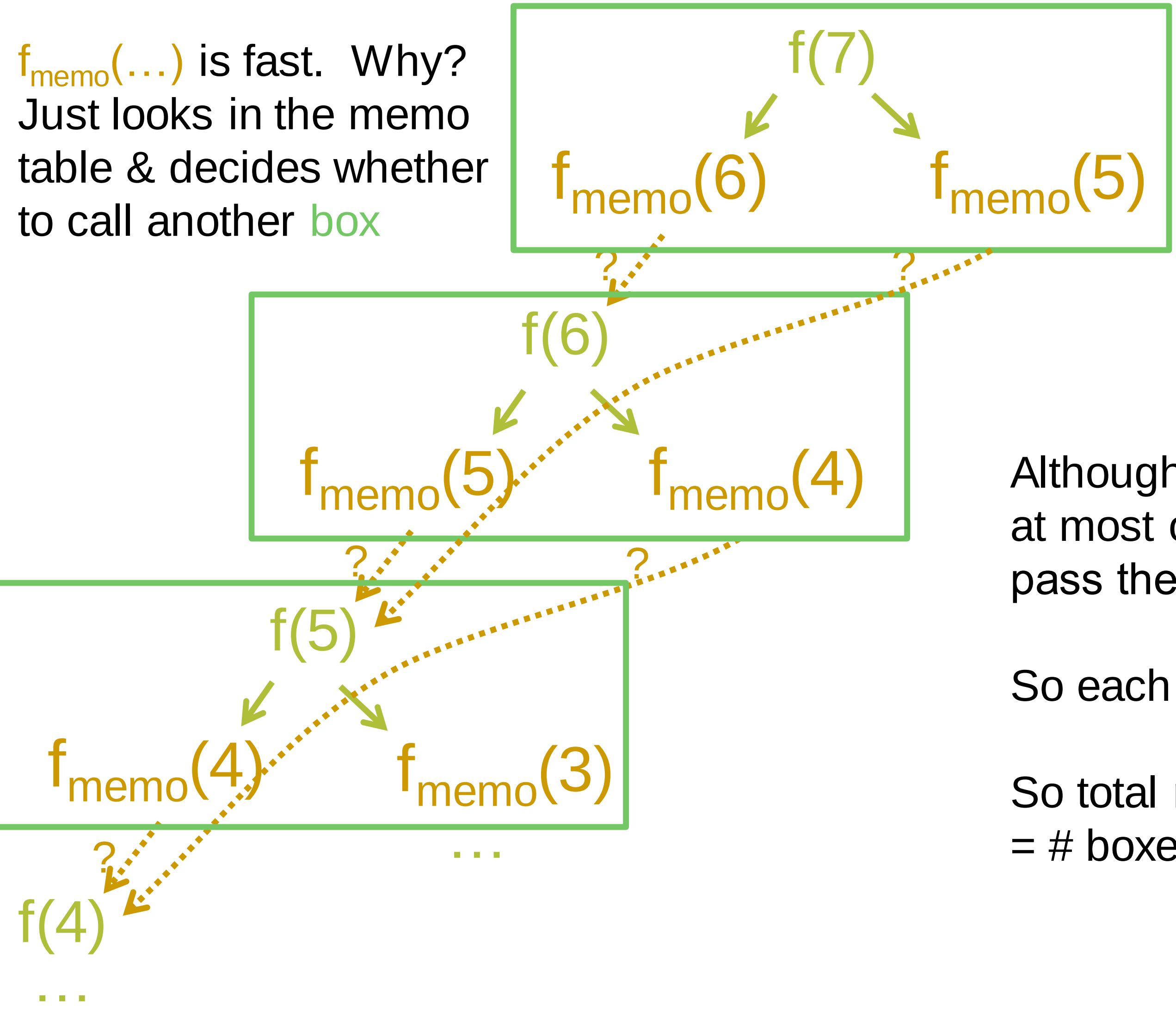
Memoization is why you only have to count each node once! Let's see …

How to analyze runtime of backward chaining



How to analyze runtime of backward chaining

$f_{memo}(\ldots)$ is fast. Why? Just looks in the memo table & decides whether to call another box



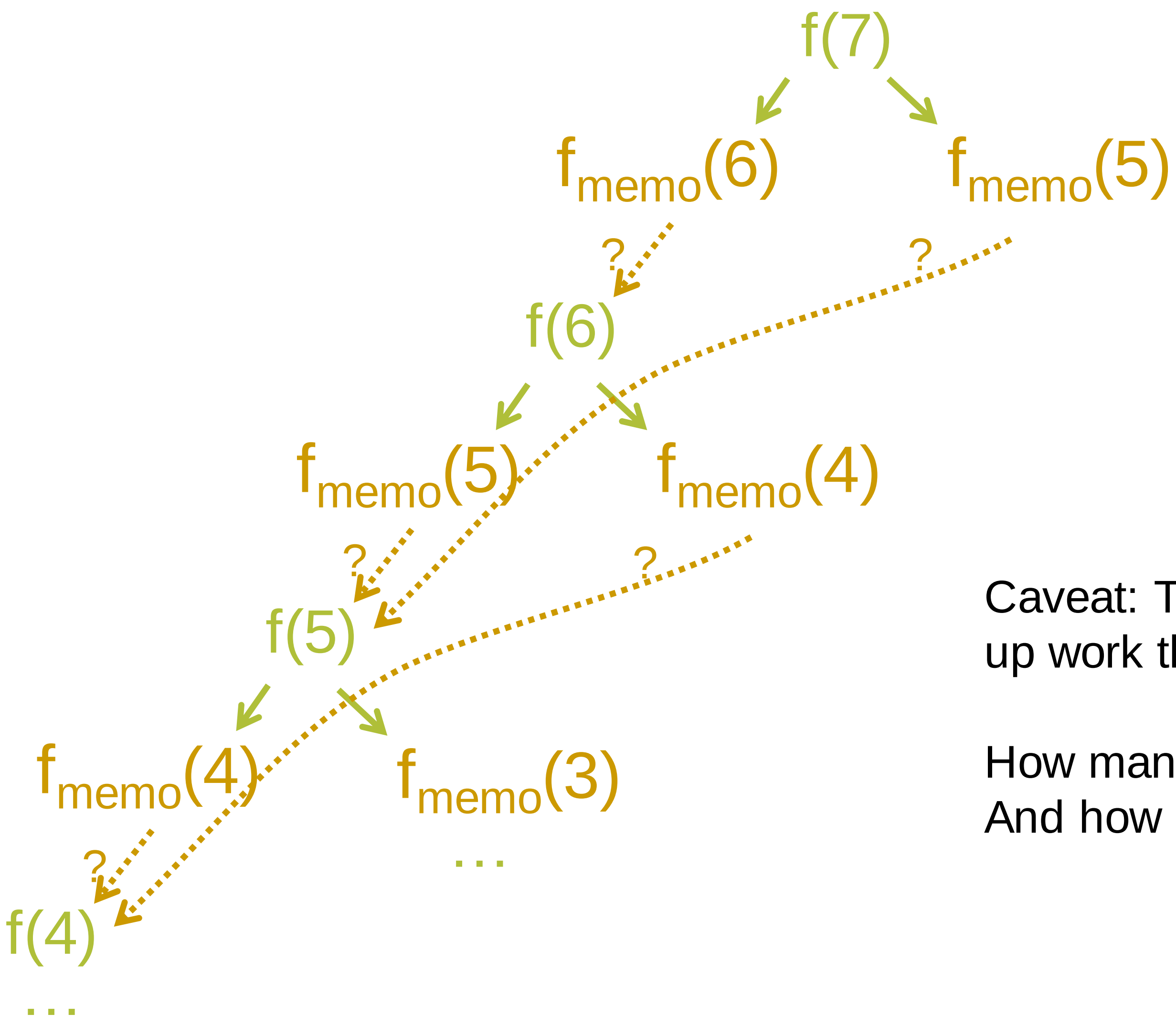
So only O(1) work *within* each box (for Fibonacci)

Although $f_{memo}(5)$ gets called twice, at most one of those calls will pass the "?" and call the f(5) box

So each box gets called only once!

So total runtime
= # boxes * average runtime per box

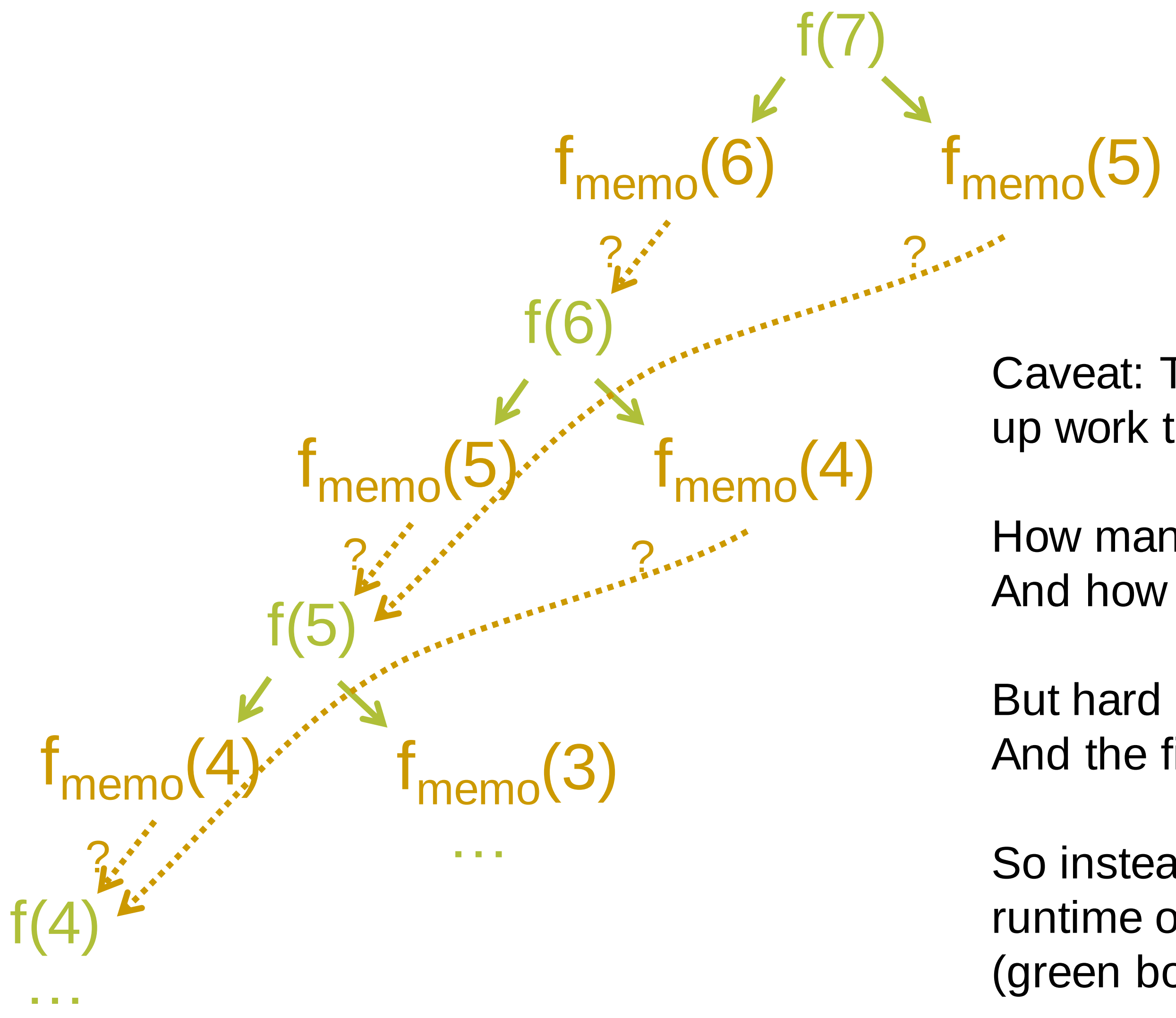
How to analyze runtime of backward chaining



Caveat: Tempting to try to divide up work this way:

How many calls to $f_{memo}(n)$?
And how long does each one take?

How to analyze runtime of backward chaining



Caveat: Tempting to try to divide up work this way:

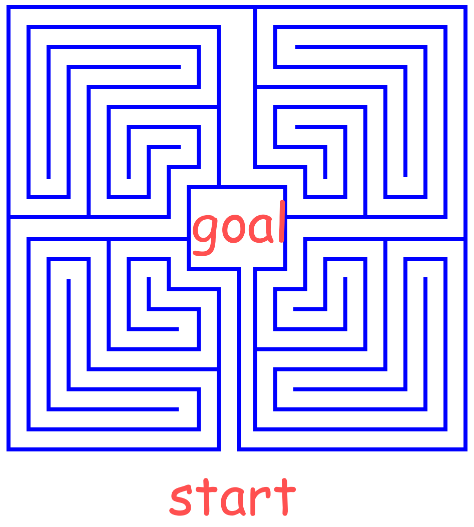
How many calls to $f_{memo}(n)$?
And how long does each one take?

But hard to figure out how many.
And the first one is slower than rest!

So instead, our previous trick associates runtime of $f_{memo}(n)$ with its *caller* (green boxes, not brown blobs)

Which direction is better in general?

- Is it easier to start at the entrance and forward-chain toward the goal, or start at the goal and work backwards?



- Depends on who designed the maze …
- In general, depends on your problem.

Another example: binomial coefficients

- Pascal's triangle

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

…

Another example: binomial coefficients

- Pascal's triangle

```
                    c(0,0)
                c(1,0)  c(1,1)
            c(2,0)  c(2,1)  c(2,2)
        c(3,0)  c(3,1)  c(3,2)  c(3,3)
    c(4,0)  c(4,1)  c(4,2)  c(4,3)  c(4,4)
                      …
```

Suppose your goal is to compute c(4,1). What is the forward-chained order?

<u>Double</u> loop this time:

```
for n=0 to 4
  for k=0 to n
    c[n,k] = …
```

c(0,0) += 1.
c(N,K) += c(N-1,K-1).
c(N,K) += c(N-1,K).

Can you save memory as in the Fibonacci example?
Can you exploit symmetry?

Another example: binomial coefficients

- Pascal's triangle

$$\begin{tikzcd}
 & & & & c(0,0) & & & & \\
 & & & c(0,1) \arrow[ur] & & c(1,1) \arrow[ul] & & & \\
 & & c(0,2) \arrow[ur] & & c(1,2) \arrow[ul] \arrow[ur] & & c(2,2) & & \\
 & c(0,3) \arrow[ur] & & c(1,3) \arrow[ul] \arrow[ur] & & c(2,3) & & c(3,3) & \\
c(0,4) & & c(1,4) \arrow[ul] \arrow[ur] & & c(2,4) & & c(3,4) & & c(4,4)
\end{tikzcd}$$

…

c(0,0) += 1.
c(N,K) += c(N-1,K-1).
c(N,K) += c(N-1,K).

Suppose your goal is to compute c(4,1). What is the backward-chained order?

Less work in this case: only compute on an as-needed basis, so actually compute less.

Figure shows importance of memoization!

But how do we stop backward or forward chaining from running forever?

Another example: Sequence partitioning

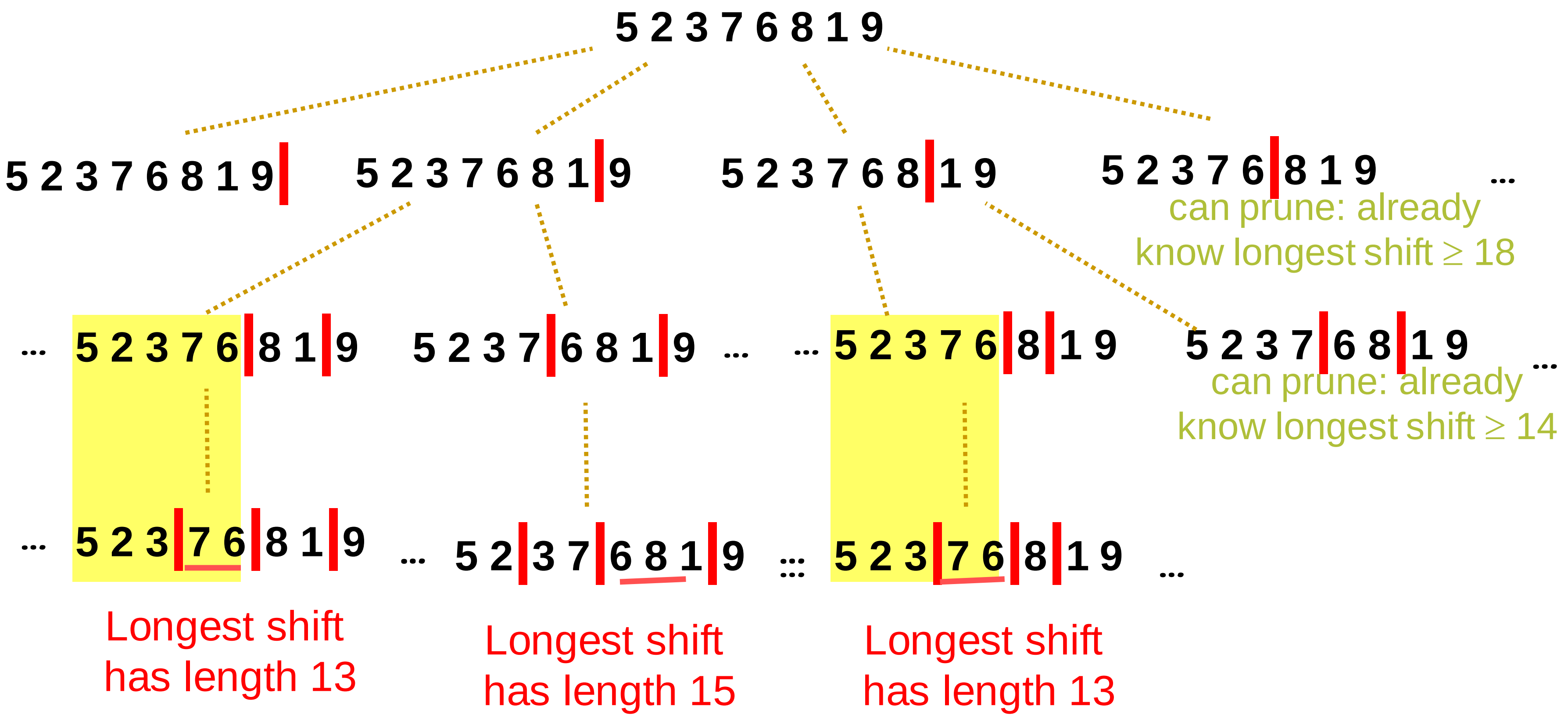
[solve in class]

- Sequence of n tasks to do <u>in order</u>
- Let amount of work per task be s_1, s_2, … s_n
- Divide into k shifts so that no shift gets too much work
  - i.e., minimize the max amount of work on any shift

Another example: Sequence partitioning

Divide sequence of n=8 tasks into k=4 shifts – need to place 3 boundaries

Branch and bound: place 3rd boundary, then 2nd, then 1st



These are really solving the **same subproblem** (n=5, k=2)
Longest shift in this subproblem = 13
So longest shift in full problem = max(13,9) or max(13,10)

Use **dynamic programming**!

Another example: Sequence partitioning

Divide sequence of N tasks into K shifts

We have a minimization problem (min worst shift).

Place last j jobs (for some j ≤ N) into last (Kth) shift, and recurse to partition earlier jobs. (Base case?)

$$f(N,K) = \min_{j=0}^{N} \Big(\max(f(N-j, K-1), \sum_{i=N-j+1}^{N} s_i) \Big)$$

- Solution at http://snipurl.com/23c2xrn
- What is the runtime? Can we improve it?

- Variant: Could use more than k shifts, but an extra cost for adding each extra shift

Another example: Sequence partitioning

Divide sequence of N tasks into K shifts

```
int f(N,K):        // memoize this!
  if K=0           // have to divide N tasks into 0 shifts
    if N=0 then return -∞ else return ∞    // impossible for N > 0
  else             // consider # of tasks in last shift
    bestanswer = ∞     // keep a running minimum here
    lastshift = 0      // total work currently in last shift
    while N ≥ 0        // number of tasks not currently in last shift
      if (lastshift < bestglobalsolution) then break    // prune node
      bestanswer  min=  max(f(N,K-1),lastshift)
      lastshift += s[N]    // move another task into last shift
      N = N-1
    return bestanswer
```

Another example: Sequence partitioning

Divide sequence of N tasks into K shifts

- Dyna version?

Another example: Knapsack problem

[solve in class]

- You're packing for a camping trip (or a heist)
  - Knapsack can carry 80 lbs.
- You have n objects of various weight and value to you
  - Which subset should you take?
  - Want to maximize total value with weight ≤ 80
- Brute-force: Consider all subsets
- Dynamic programming:
  - Pick an arbitrary order for the objects (like variable ordering!)
    - weights $w_1, w_2, \ldots w_n$ and values $v_1, v_2, \ldots v_n$
  - Let c[i,w] be max value of any subset of the first i items (only) that weighs ≤ w pounds

Knapsack problem is NP-complete

- What's the runtime of algorithm below? Isn't it polynomial?
- The problem: What if w is a 300-bit number?
  - Short encoding , but the w factor is very large (2^{300})
  - How many different w values will actually be needed if we compute "as needed" (backward chaining + memoization)?
- Dynamic programming:
  - Pick an arbitrary order for the objects
    - weights $w_1, w_2, \ldots w_n$ and values $v_1, v_2, \ldots v_n$
  - Let c[i,w] be max value of any subset of the first i items (only) that weighs $\leq$ w pounds

Might be better when w large:
Let d[i,v] be min weight of any subset of the first i items (only) that has value $\geq$ v

The problem of redoing work

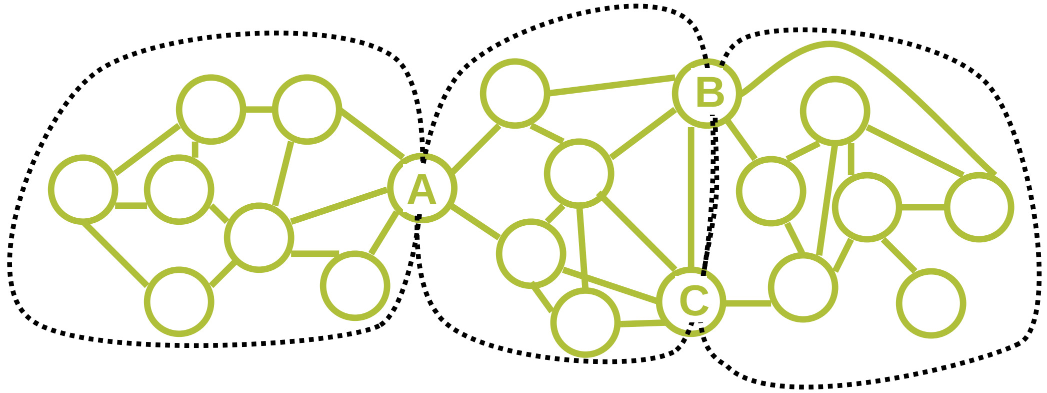
- Note: We've seen this before. A major issue in SAT/constraint solving – try to figure out automatically how to avoid redoing work.

- Let's go back to graph coloring for a moment.
  - Moore's animations #3 and #8
  - http://www-2.cs.cmu.edu/~awm/animations/constraint/

- What techniques did we look at?
  - Clause learning or backjumping (like memoization!):
    - "If v5 is black, you will always fail."
    - "If v5 is black or blue or red, you will always fail" (so give up!)
    - "If v5 is black then v7 must be blue and v10 must be red or blue …"

The problem of redoing work

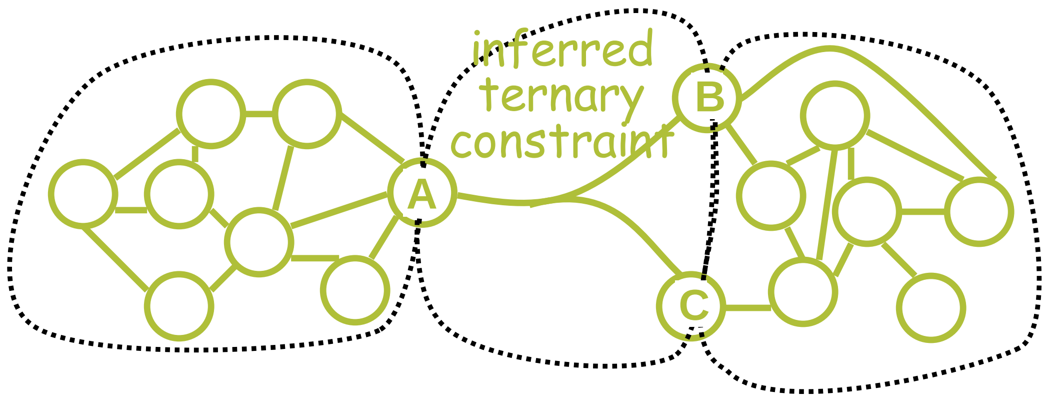
- Note: We've seen this before. A major issue in SAT/constraint solving: try to figure out automatically how to avoid redoing work.
- Another strategy, inspired by dynamic programming:
  - Divide graph into subgraphs that touch only occasionally, at their peripheries.
  - Recursively solve these subproblems; store & reuse their solutions.



  - Solve each subgraph first. What does this mean?
    - What combinations of colors are okay for (A,B,C)?
    - That is, join subgraph's constraints and project onto its periphery.
  - How does this help when solving the main problem?

The problem of redoing work

- Note: We've seen this before. A major issue in SAT/constraint solving: try to figure out automatically how to avoid redoing work.
- Another strategy, inspired by dynamic programming:
  - Divide graph into subgraphs that touch only occasionally, at their peripheries.
  - Recursively solve these subproblems; store & reuse their solutions.



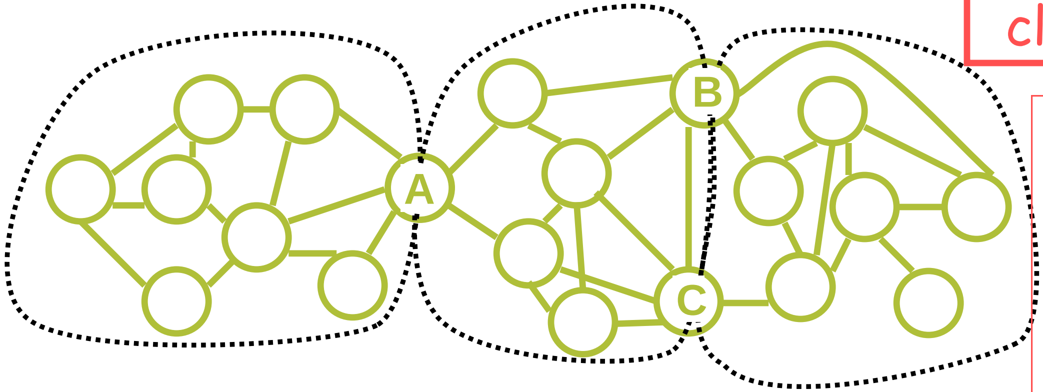
Variable ordering, variable (bucket) elim., and clause learning are basically hoping to find such a decomposition.

  - Solve each subgraph first. What does this mean?
    - What combinations of colors are okay for (A,B,C)?
    - That is, join subgraph's constraints and project onto its periphery.
  - **How does this help when solving the main problem?**

The problem of redoing work

- Note: We've seen this before. A major issue in SAT/constraint solving: try to figure out <u>automatically</u> how to avoid redoing work.
- Another strategy, inspired by dynamic programming:
  - Divide graph into subgraphs that touch only occasionally, at their peripheries.
  - Recursively solve these subproblems; $\boxed{\text{store \& reuse their solutions.}}$

clause learning on A,B,C



To join constraints in a subgraph:
Recursively solve subgraph by backtracking, variable elimination, …

  - Solve each subgraph first:
    - What combinations of colors are okay for (A,B,C)?
    - That is, <u>join</u> subgraph's constraints and project onto its periphery.
  - How does this help when solving the main problem?

The problem of redoing work

- Note: We've seen this before. A major issue in SAT/constraint solving: try to figure out <u>automatically</u> how to avoid redoing work.
- Another strategy, inspired by dynamic programming:
  - Divide graph into subgraphs that touch only occasionally, at their peripheries.

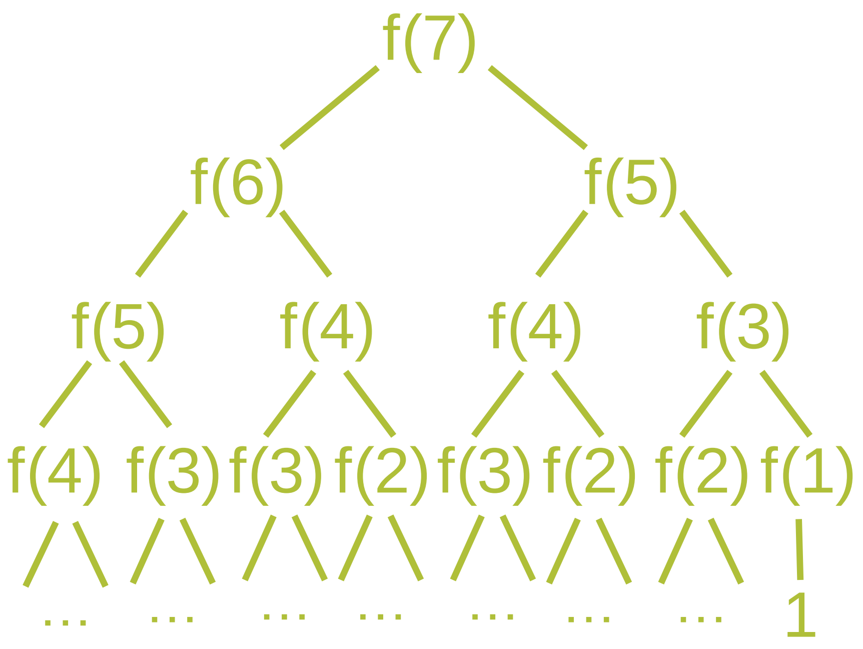
- Dynamic programming usually means dividing your problem up <u>manually</u> in some way.

- Break it into smaller subproblems.
- $\boxed{\text{Solve them first}}$ and combine the subsolutions.
- Store the subsolutions for multiple re-use.

Because a recursive call specifically told you to (backward chaining), or because a loop is solving <u>all</u> smaller subproblems (forward chaining).

Fibonacci series

```
int f(int n) {
    if n < 2
        return n
    else
        return f(n-1) + f(n-2)
}
```



So is the problem really only about the fact that we recurse <u>twice?</u>
Yes – why can we get away without DP if we only recurse once?
Is it common to recurse more than once?

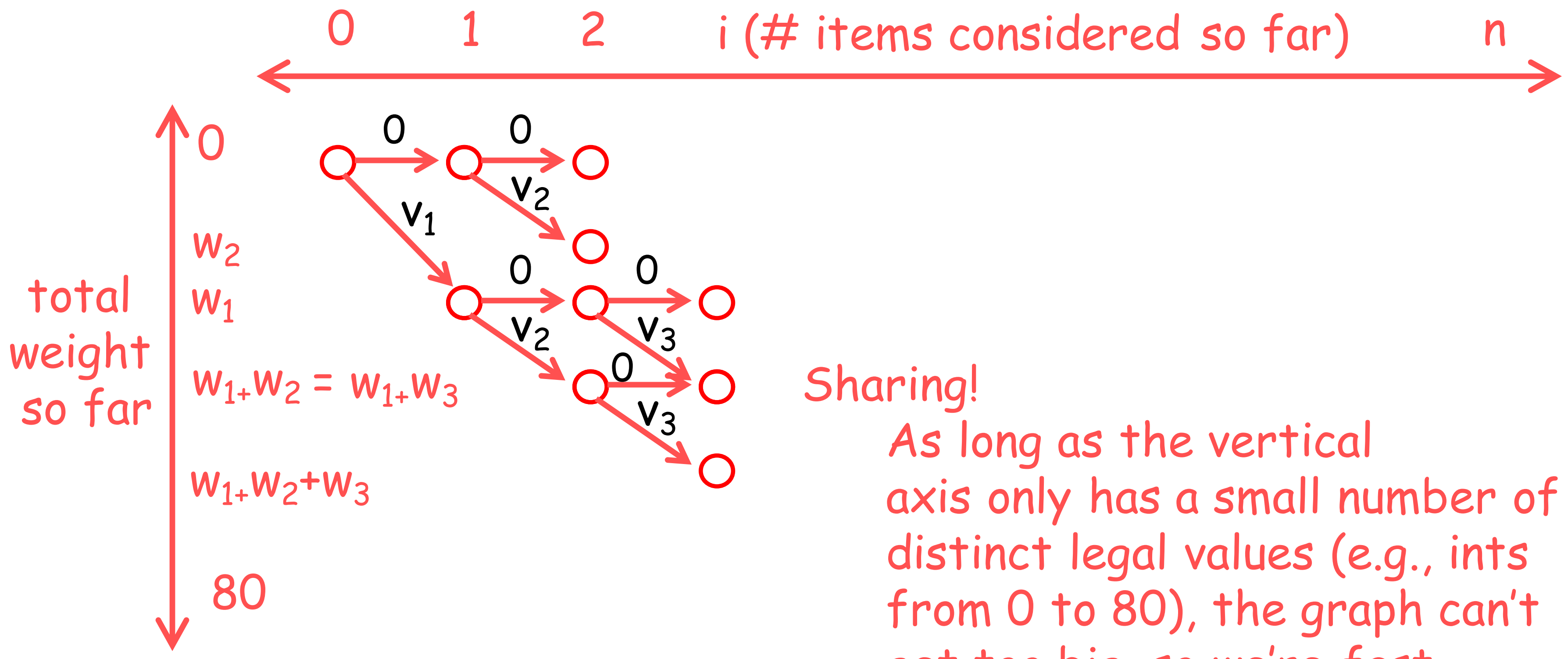
Sure! Whenever we try multiple ways to solve the problem to see if any solution exists, or to pick the best solution. Ever hear of backtracking search? How about Prolog?

Many dynamic programming problems = shortest path problems

- Not true for Fibonacci, or game tree analysis, or natural language parsing, or …
- But true for knapsack problem and others.
- Let’s reduce knapsack to shortest path!

Many dynamic programming problems = shortest path problems

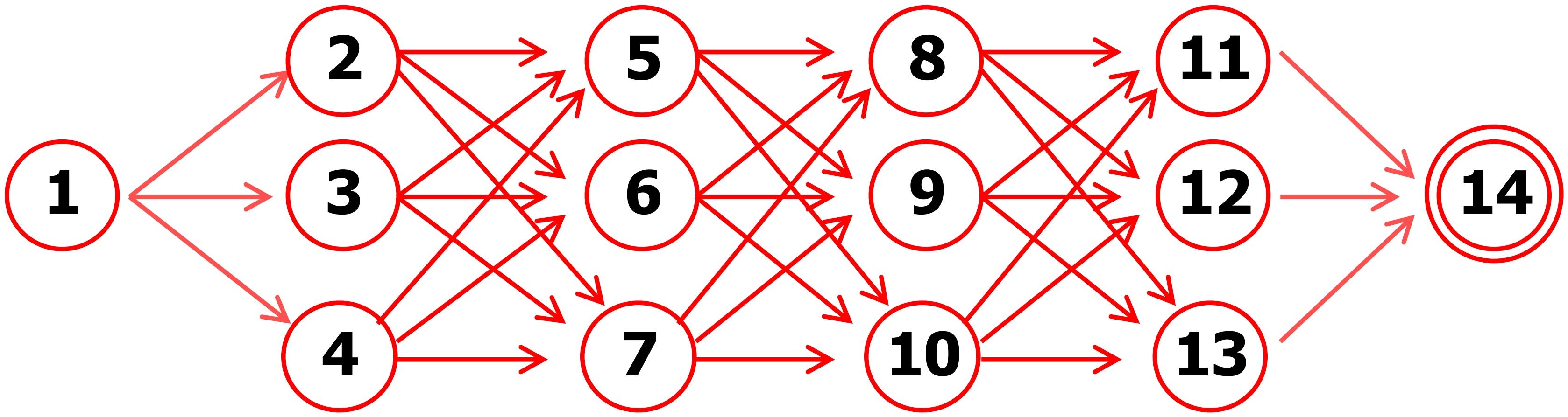
- Let's reduce knapsack to ~~shortest~~ longest path!



Sharing!
As long as the vertical axis only has a small number of distinct legal values (e.g., ints from 0 to 80), the graph can't get too big, so we're fast.

Path-finding in Prolog

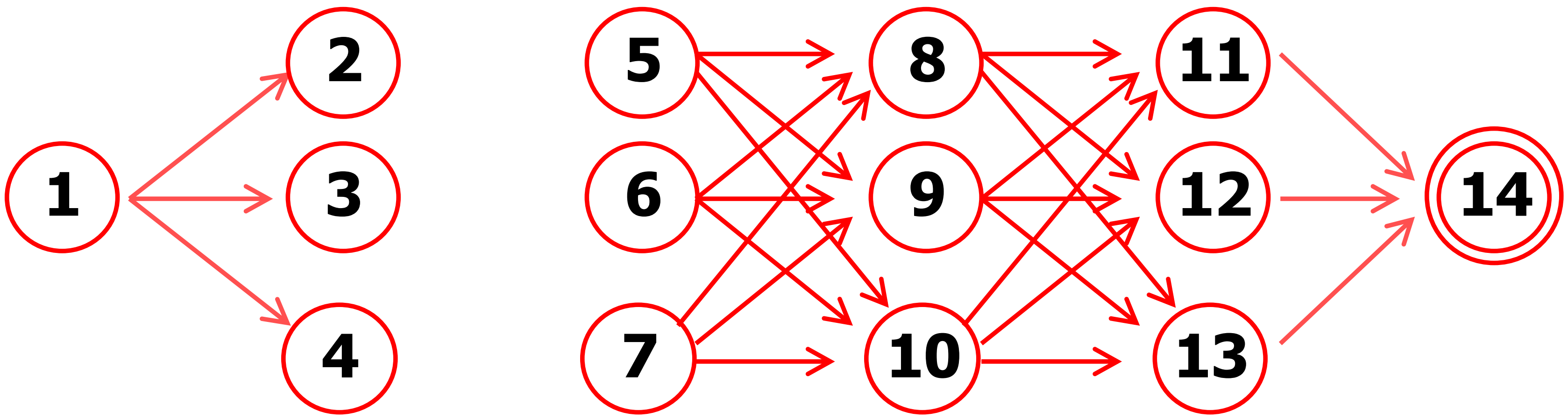
- pathto(1). % the start of all paths
  pathto(V) :- edge(U,V), pathto(U).
- When is the query pathto(14) really inefficient?



- What does the recursion tree look like? (very branchy)
- What if you merge its nodes, using memoization?
  - (like the picture above, turned sideways ☺)

Path-finding in Prolog

- pathto(1).   % the start of all paths
  pathto(V) :- edge(U,V), pathto(U).



- Forward vs. backward chaining? (Really just a maze!)
- How about cycles?
- How about weighted paths?

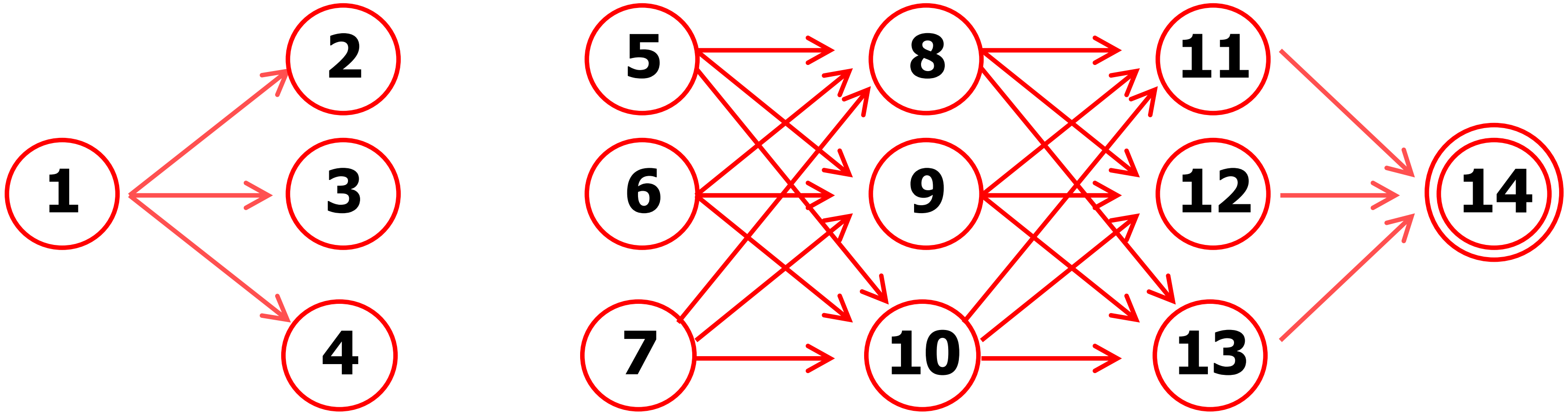
Path-finding in **Dyna**

solver uses dynamic programming for efficiency

- pathto(1) = true.
  pathto(V) |= edge(U,V) & pathto(U).

In Dyna, okay to swap order …

Recursive formulas on booleans.



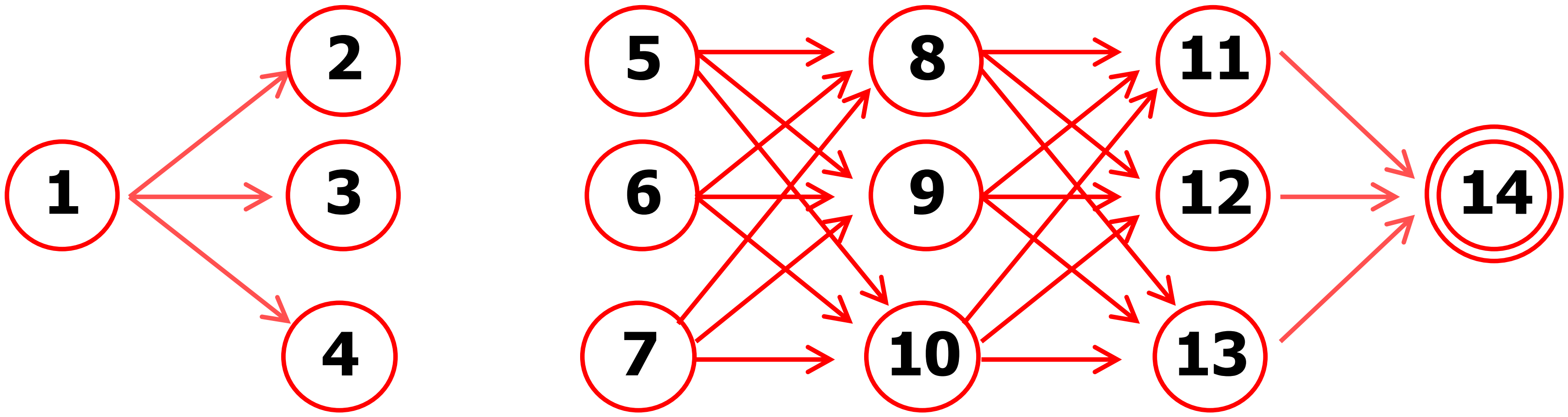
Path-finding in **Dyna**

solver uses dynamic programming for efficiency

- pathto(1) = true.
  pathto(V) |= pathto(U) & edge(U,V).

In Dyna, okay to swap order …

Recursive formulas on booleans.



1. pathto(V) min= pathto(U) + edge(U,V).
2. pathto(V) max= pathto(U) * edge(U,V).
3. pathto(V) += pathto(U) * edge(U,V).

3 weighted versions: Recursive formulas on real numbers.

Path-finding in Dyna

solver uses dynamic programming for efficiency

- pathto(V) min= pathto(U) + edge(U,V).
  - "Length of shortest path from Start?"
  - For each vertex V, pathto(V) is the minimum over all U of pathto(U) + edge(U,V).
- pathto(V) max= pathto(U) * edge(U,V).
  - "Probability of most probable path from Start?"
  - For each vertex V, pathto(V) is the maximum over all U of pathto(U) * edge(U,V).
- pathto(V) += pathto(U) * edge(U,V).
  - "Total probability of all paths from Start (maybe ∞ly many)?"
  - For each vertex V, pathto(V) is the sum over all U of pathto(U) * edge(U,V).
- pathto(V) |= pathto(U) & edge(U,V).
  - "Is there a path from Start?"
  - For each vertex V, pathto(V) is true if there exists a U such that pathto(U) and edge(U,V) are true.

The Dyna project

- Dyna is a language for <u>computation</u>.
- It’s especially good at dynamic programming.
- Differences from Prolog:
  - More powerful – values, aggregation (min=, +=)
  - Faster solver – dynamic programming, etc.

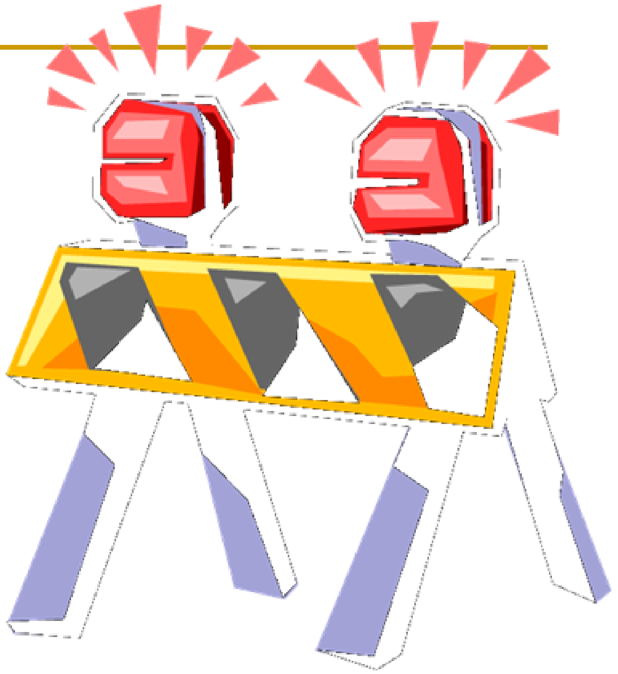
- We’re developing it here at JHU CS.
- Makes it <u>much</u> faster to build our NLP/ML systems.
- You may know someone working on it.
  - Great hackers welcome

The Dyna project

- Insight:
  - Many algorithms are fundamentally based on a set of equations that relate some values. Those equations guarantee correctness.
- Approach:
  - Who really cares what order you <u>compute</u> the values in?
  - Or what clever data structures you use to <u>store</u> them?
  - Those are mere efficiency issues.
  - Let the programmer stick to specifying the equations.
  - Leave efficiency to the compiler.
- Question for next week:
  - The compiler has to know good tricks, like any solver.
  - So what are the key solution techniques for dynamic programming?

- Please read http://www.dyna.org/Several_perspectives_on_Dyna

Not everything works yet



- We're currently designing & building Dyna 2.
- We do have two earlier prototypes:
  - https://github.com/nwf/dyna (friendlier)
  - http://dyna.org (earlier, faster, more limited)
- Overview talk on the language (slides + video):
  - http://cs.jhu.edu/~jason/papers/#eisner-2009-ilpmlgsrl

Fibonacci

- fib(z) = 0.
- fib(s(z)) = 1.
- fib(s(s(N))) = fib(N) + fib(s(N)).

- If you use := instead of = on the first two lines, you can change 0 and 1 at runtime and watch the changes percolate through:
  3, 4, 7, 11, 18, 29, …

Fibonacci

- fib(s(z)) = 1.
- fib(s(N))) += fib(N).
- fib(s(s(N))) += fib(N).

Fibonacci

Good for forward chaining:
have fib(M), let N=M+1, add to fib(N)

- fib(1) = 1.
- fib(M+1) += fib(M).
- fib(M+2) += fib(M).

Note: M+1 is evaluated in place, so fib(6+1) is equivalent to fib(7).
Whereas in Prolog it would just be the nested term fib('+'(6,1)).

Fibonacci

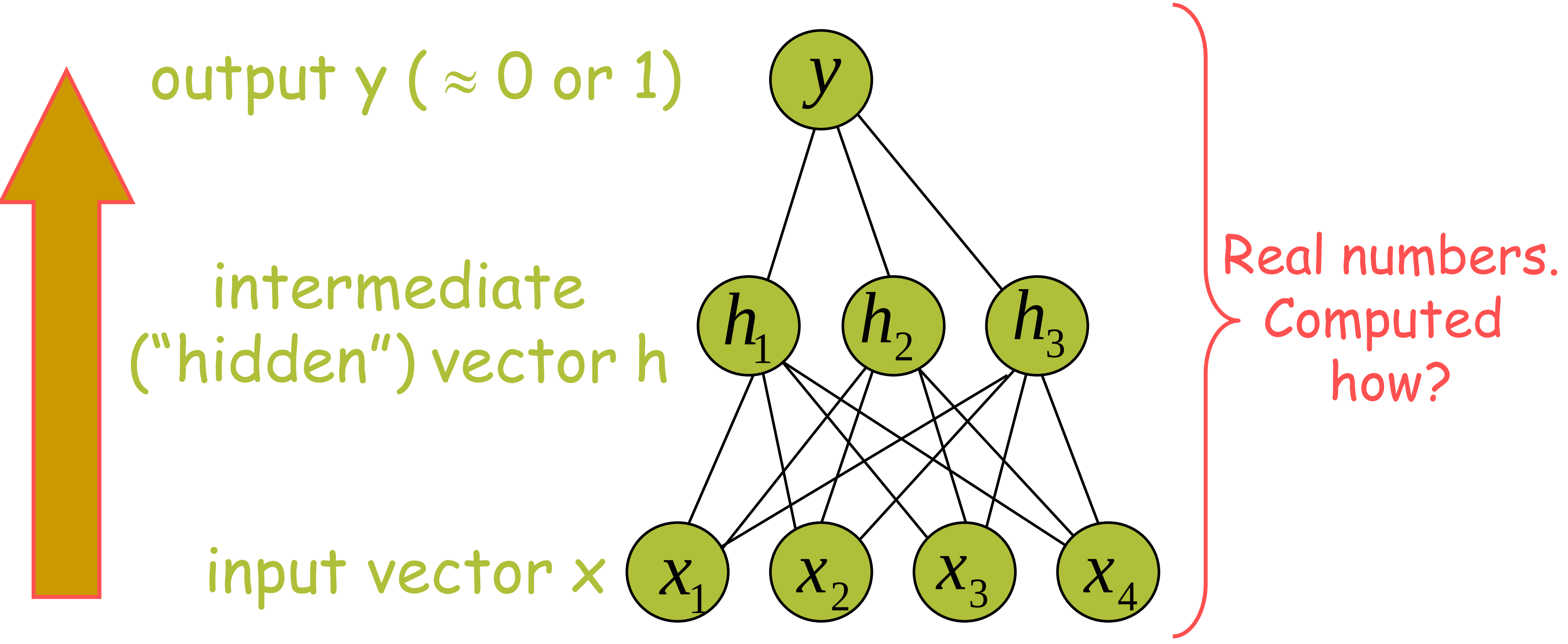
Good for backward chaining:
want fib(N), let M=N-1, request fib(M)

- fib(1) = 1.
- fib(N) += fib(N-1).
- fib(N) += fib(N-2).

Note: N-1 is evaluated in place, so fib(7-1) is equivalent to fib(6).
Whereas in Prolog it would just be the nested term fib('-'(7,1)).

Architecture of a neural network

(a basic "multi-layer perceptron" – there are other kinds)

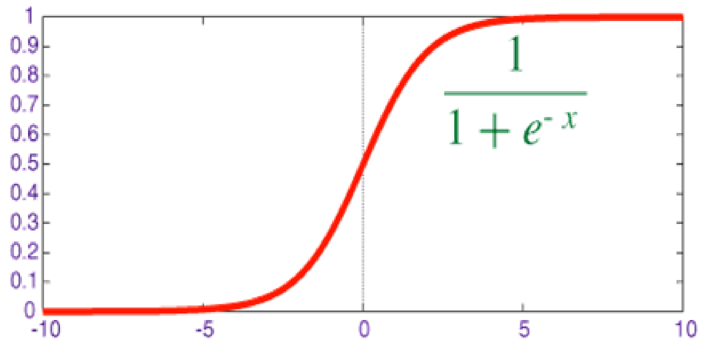


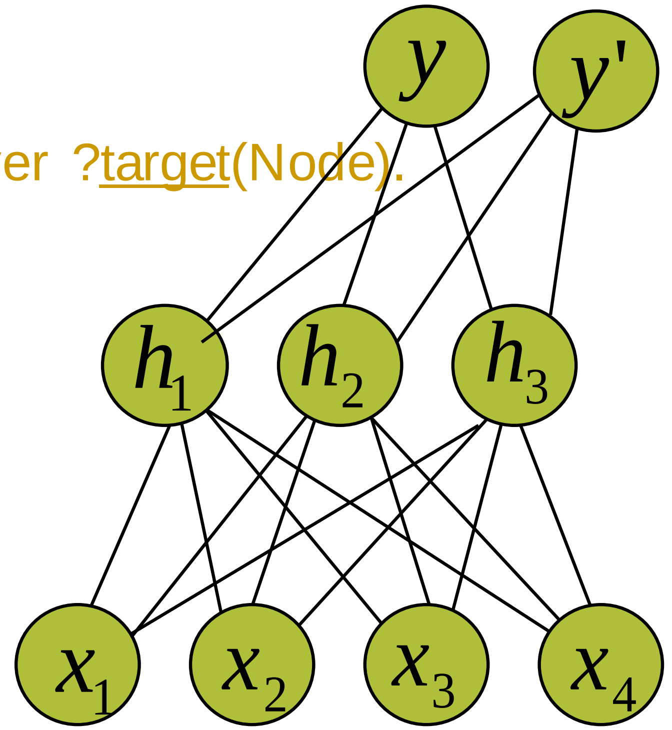
Small example … often x and h are much longer vectors

Neural networks in Dyna

- in(Node) += weight(Node,Previous)*out(Previous).
- in(Node) += input(Node).
- out(Node) = sigmoid(in(Node)).
- error += (out(Node)-target(Node))**2 whenever ?target(Node).

- :- foreign(sigmoid). % defined in C++





- What are the initial facts ("axioms")?
- Should they be specified at compile time or runtime?
- How about training the weights to minimize error?
- Are we usefully storing partial results for reuse?

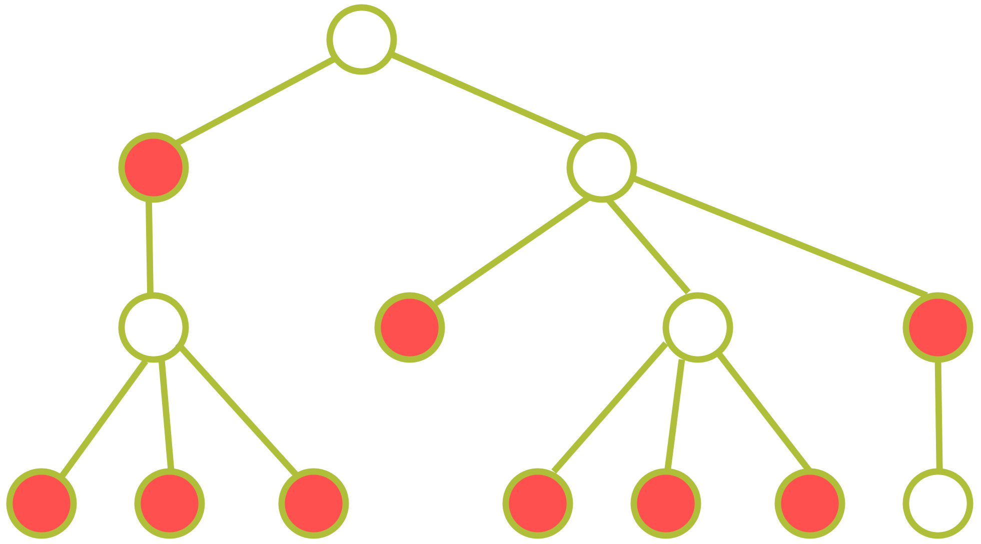
Maximum independent set in a tree

- A set of vertices in a graph is “independent” if no two are neighbors.

- In general, finding a <u>maximum-size</u> independent set in a graph is NP-complete.
- But we can find one in a tree, using dynamic programming …
- This is a typical kind of problem.

Maximum independent set in a tree

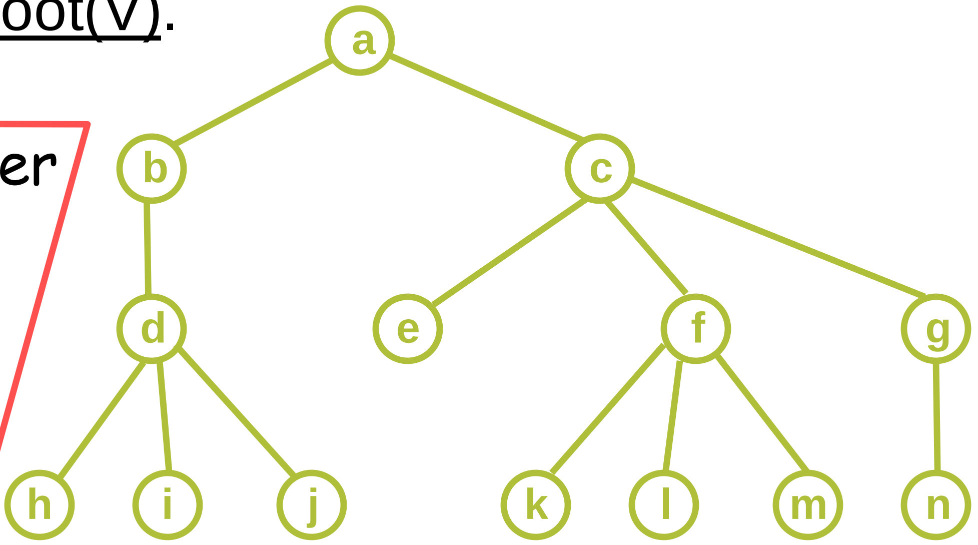
- Remember: A set of vertices in a graph is "independent" if no two are neighbors.
- **Think about how to find max indep set …**



Silly application:
Get as many members of this family on the corporate board as we can, subject to law that parent & child can't serve on the same board.

How do we represent our tree in Dyna?

- One easy way: represent the tree like any graph.
  parent("a", "b"). parent("a", "c"). parent("b", "d"). …
- To get the size of the subtree rooted at vertex V:
  size(V) += 1. % root
  size(V) += size(Kid) whenever <u>parent(V,Kid)</u>. % children
- Now to get the total size of the whole tree,
  goal += size(V) whenever <u>root(V)</u>.
  root("a").
- This finds the total number of members that could sit on the board <u>if there were no parent/child law.</u>
- How do we fix it to find max <u>independent</u> set?

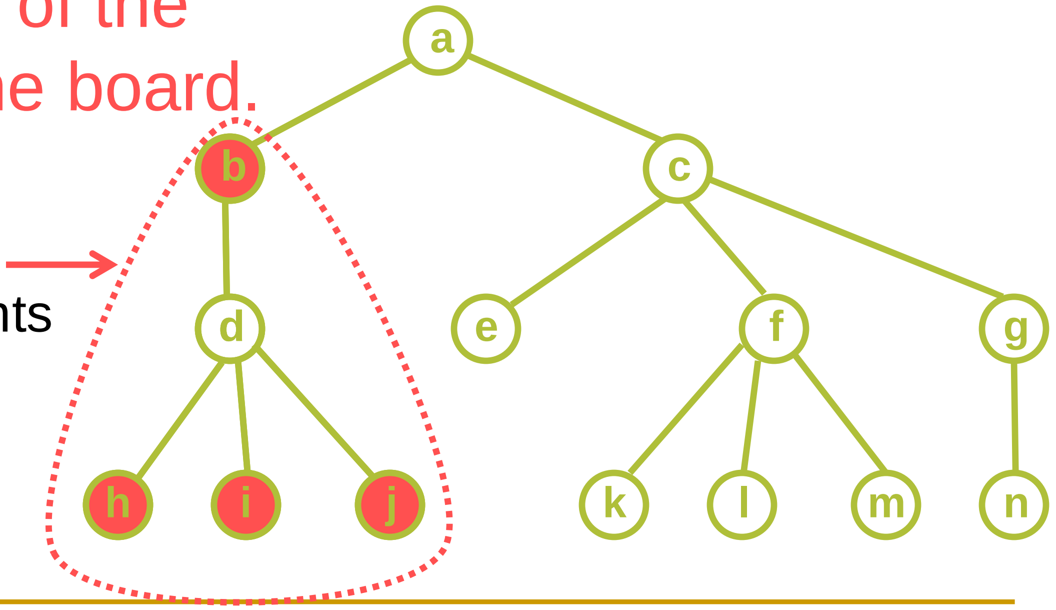


Maximum independent set in a tree

- Want the maximum independent set rooted at a.
- It is not enough to solve this for a's two child subtrees. Why not?

- Well, okay, turns out that actually it is enough. ☺
- So let's go to a slightly harder problem:
- Maximize the total IQ of the family members on the board.

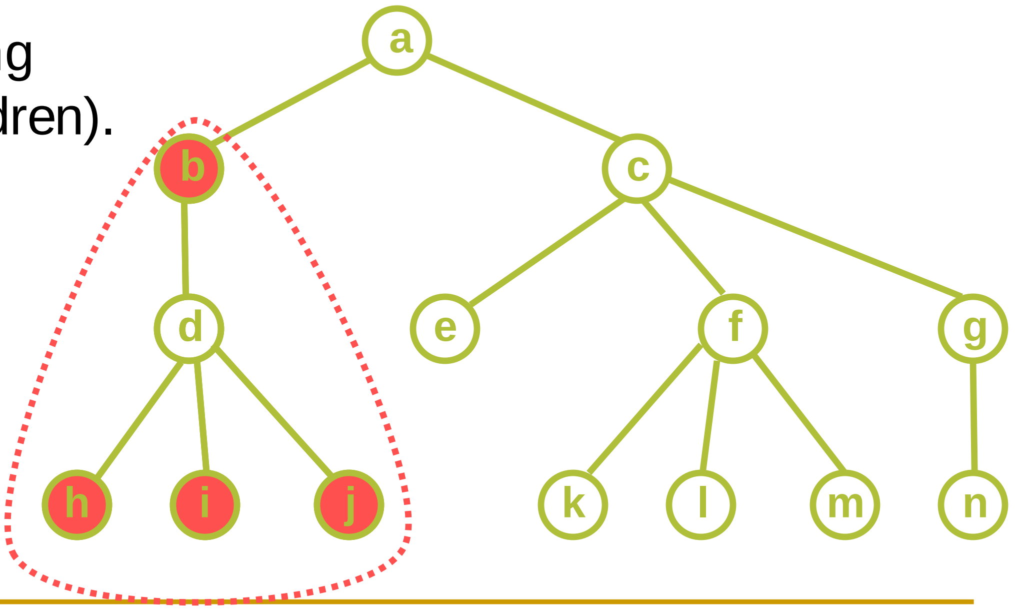
- This is the best solution for the left subtree, but it prevents "a" being on the board.
- So it's a bad idea if "a" has an IQ of 2,000.



Treating it as a MAX-SAT problem

- Hmm, we could treat this as a MAX-SAT problem. Each vertex is T or F according to whether it is in the independent set.
- What are the hard constraints (legal requirements)?
- What are the soft constraints (our preferences)? Their weights?

- What does backtracking search do?
- Try a top-down variable ordering (assign a parent before its children).
- What does unit propagation now do for us?
- Does it prevent us from taking exponential time?
- We must try c=F twice: for both a=T and a=F.

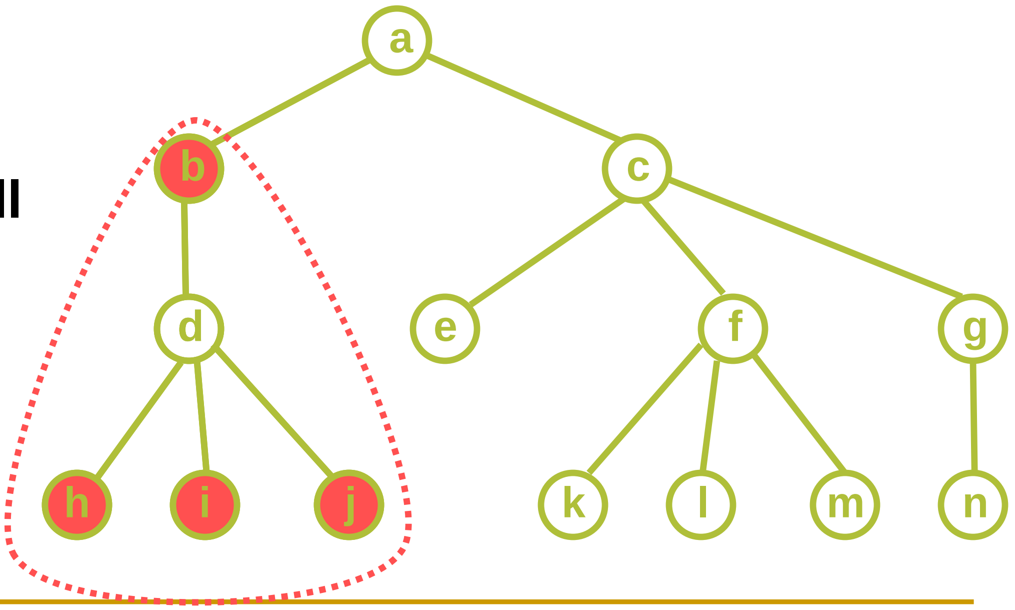


Same point upside-down …

- We could also try a bottom-up variable ordering.
- You might write it that way in Prolog:
- For each satisfying assignment of the left subtree,

```
for each satisfying assignment of the right subtree,
    for each consistent value of root (F and maybe T),
        Benefit = total IQ.    % maximize this
```

- But to determine whether T is consistent at the root a, do we really care about the **full** satisfying assignment of the left and right subtrees?
- No! We only care about the **roots** of those solutions (b, c).



Maximum independent set in a tree

- Enough to find a subtree's best solutions for root=T and for root=F.
- Break up the "size" predicate as follows:

  any(V) = size of the max independent set in the subtree rooted at V

  rooted(V) = like any(V), but only considers sets that include V itself

  unrooted(V) = like any(V), but only considers sets that exclude V itself

- any(V) = rooted(V) max unrooted(V). % whichever is bigger

- rooted(V) += iq(V). % intelligence quotient
- rooted(V) += unrooted(Kid) whenever parent(V,Kid).

} V=T case. uses unrooted(Kid).

- unrooted(V) += any(Kid) whenever parent(V,Kid).

V=F case. uses rooted(Kid) and indirectly **reuses** unrooted(Kid)!

Maximum independent set in a tree

- Problem: This Dyna program won't currently compile!
- For complicated reasons (maybe next week), you can write

  X max= Y + Z (also X max= Y*Z, X += Y*Z, X |= Y & Z …)

  but not

  X += Y max Z
- So I'll show you an alternative solution that is also "more like Prolog."

```
any(V) = rooted(V) max unrooted(V).   % whichever is bigger

rooted(V) += iq(V).
rooted(V) += unrooted(Kid) whenever parent(V,Kid).
```

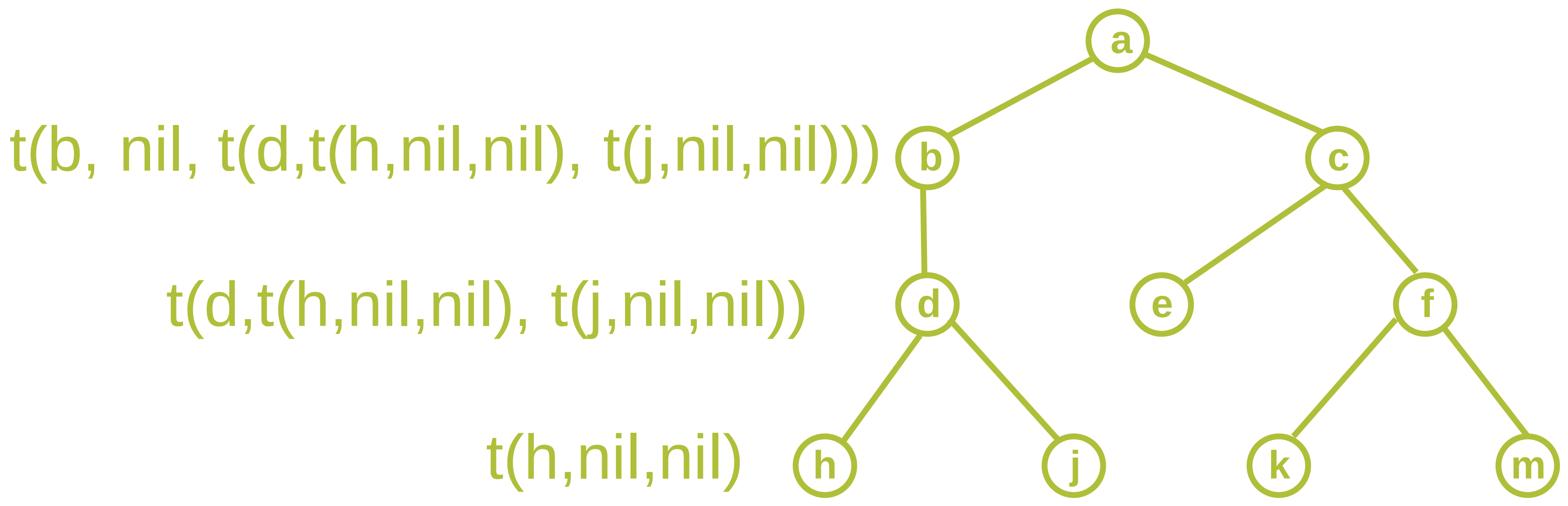
V=T case.
uses unrooted(Kid).

```
unrooted(V) += any(Kid) whenever parent(V,Kid).
```

V=F case.
uses rooted(Kid)
and indirectly **reuses** unrooted(Kid)!

A different way to represent a tree in Dyna

- Tree as a single big term
- Let's start with binary trees only:
    - t(label, subtree1, subtree2)


t(b, nil, t(d,t(h,nil,nil), t(j,nil,nil)))
t(d,t(h,nil,nil), t(j,nil,nil))
t(h,nil,nil)
a
b
c
d
e
f
h
j
k
m

Maximum independent set in a binary tree

- any(T) = the size of the maximum independent set in T
- rooted(T) = the size of the maximum independent set in T that includes T's root
- unrooted(T) = the size of the maximum independent set in T that excludes T's root

- rooted(t(R,T1,T2)) = iq(R) + unrooted(T1) + unrooted(T2).
- unrooted(t(R,T1,T2)) = any(T1) + any(T2).

- any(T) max= rooted(T).   any(T) max= unrooted(T).

- unrooted(nil) = 0.

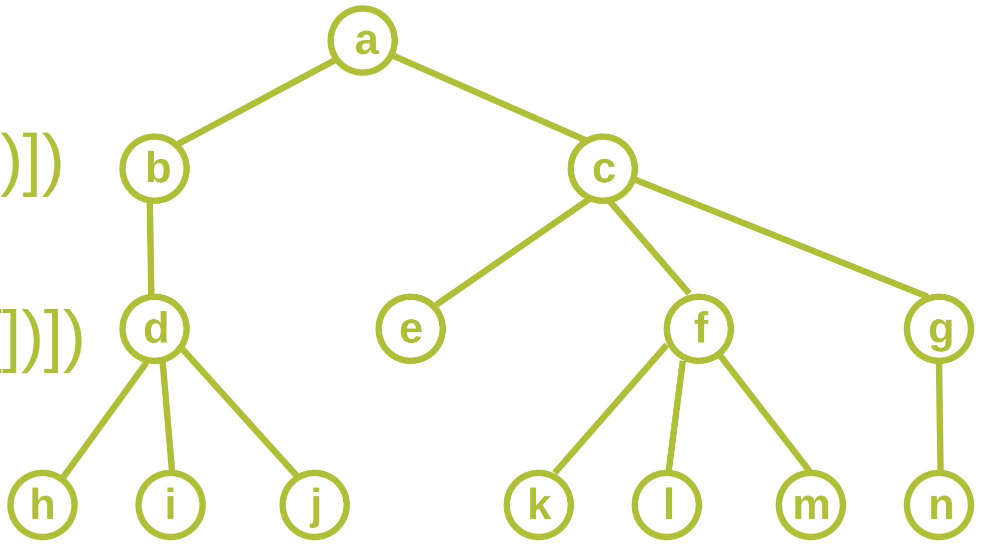
Representing arbitrary trees in Dyna

- Now let's go up to more than binary:
  - ~~t(label, subtree1, subtree2)~~
  - t(label, [subtree1, subtree2, …]).

t(b,[t(d,[t(h,[]),t(i,[]), t(j,[])])])

t(d,[t(h,[]),t(i,[]), t(j,[])])

t(h,[])



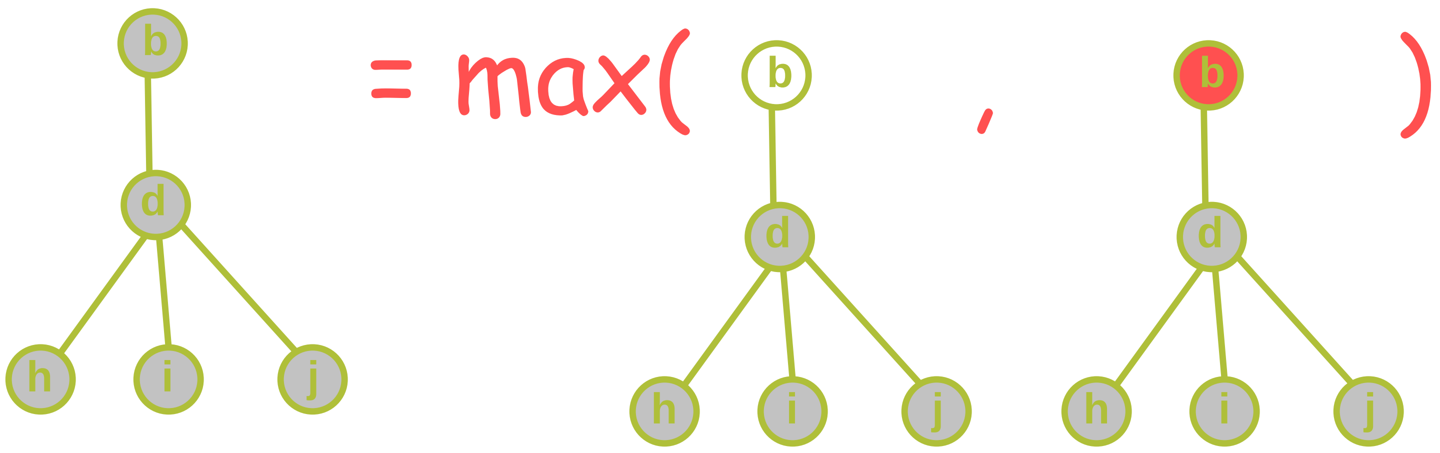
Maximum independent set in a tree

- any(T) = the size of the maximum independent set in T
- rooted(T) = the size of the maximum independent set in T that includes T's root
- unrooted(T) = the size of the maximum independent set in T that excludes T's root

as before

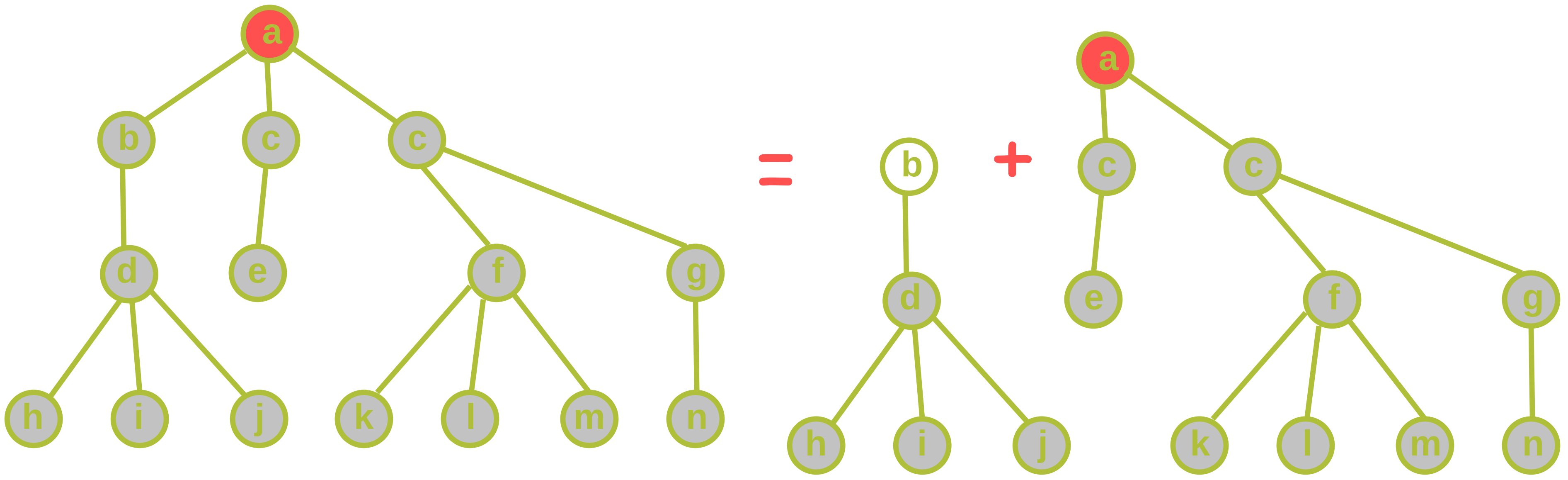
- rooted(t(R,[])) = iq(R).   unrooted(t(_,[])) = 0.
- any(T) max= rooted(T).   any(T) max= unrooted(T).
- rooted(t(R,[X|Xs])) = unrooted(X) + rooted(t(R,Xs)).
- unrooted(t(R,[X|Xs])) = any(X) + unrooted(t(R,Xs)).

Maximum independent set in a tree



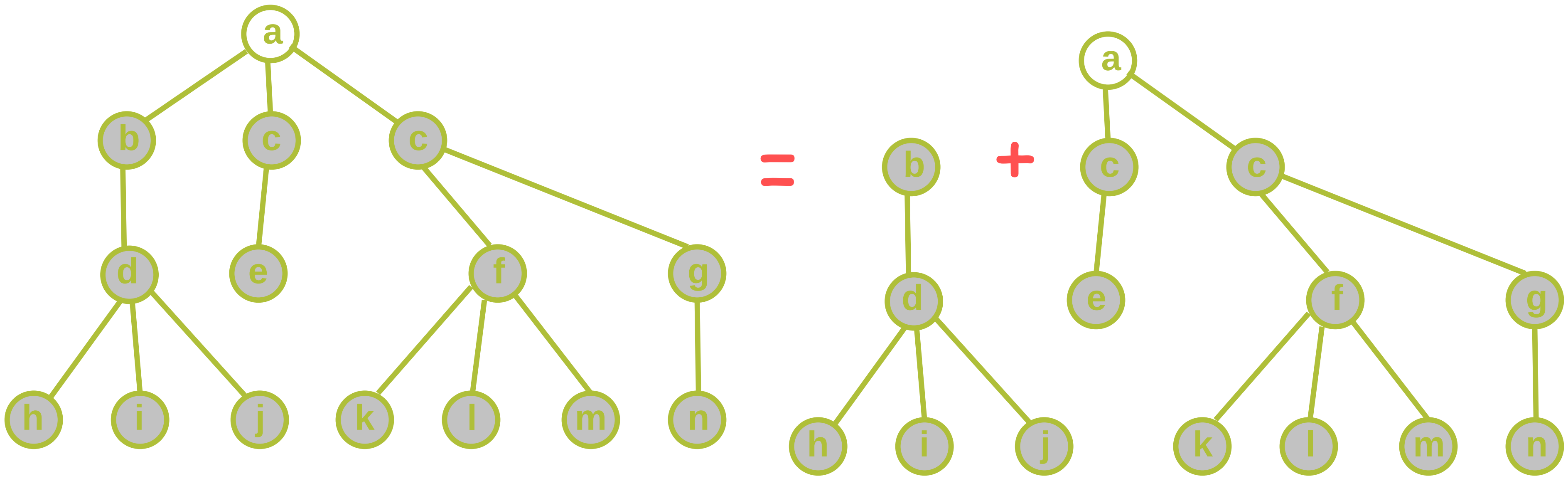
- rooted(t(R,[])) = iq(R).   unrooted(t(_,[])) = 0.
- **any(T) max= rooted(T).   any(T) max= unrooted(T).**
- rooted(t(R,[X|Xs])) = unrooted(X) + rooted(t(R,Xs)).
- unrooted(t(R,[X|Xs])) = any(X) + unrooted(t(R,Xs)).

Maximum independent set in a tree



- **rooted(t(R,[])) = <u>iq(R)</u>.** unrooted(t(_,[])) = 0.
- any(T) max= rooted(T). any(T) max= unrooted(T).
- **rooted(t(R,[X|Xs])) = unrooted(X) + rooted(t(R,Xs)).**
- unrooted(t(R,[X|Xs])) = any(X) + unrooted(t(R,Xs)).

Maximum independent set in a tree



- rooted(t(R,[])) = iq(R). **unrooted(t(_,[])) = 0.**
- any(T) max= rooted(T). any(T) max= unrooted(T).
- rooted(t(R,[X|Xs])) = unrooted(X) + rooted(t(R,Xs)).
- **unrooted(t(R,[X|Xs])) = any(X) + unrooted(t(R,Xs)).**

Maximum independent set in a tree

(shorter but harder to understand version: find it automatically?)

- We could actually eliminate "rooted" from the program. Just do everything with "unrooted" and "any."
- Slightly more efficient, but harder to convince yourself it's right.
- That is, it's an optimized version of the previous slide!

```
any(t(R,[])) = iq(R).              unrooted(t(_,[])) = 0.

any(T) max= unrooted(T).

any(t(R,[X|Xs])) = any(t(R,Xs)) + unrooted(X).
unrooted(t(R,[X|Xs])) = unrooted(t(R,Xs)) + any(X).
```

Forward-chaining would build *all* trees ☹

- rooted(t(R,[])) = iq(R).   unrooted(t(_,[])) = 0.
- any(T) max= rooted(T).   any(T) max= unrooted(T).
- rooted(t(R,[X|Xs])) = unrooted(X) + rooted(t(R,Xs)).
- unrooted(t(R,[X|Xs])) = any(X) + unrooted(t(R,Xs)).

- But which trees do we need to build?

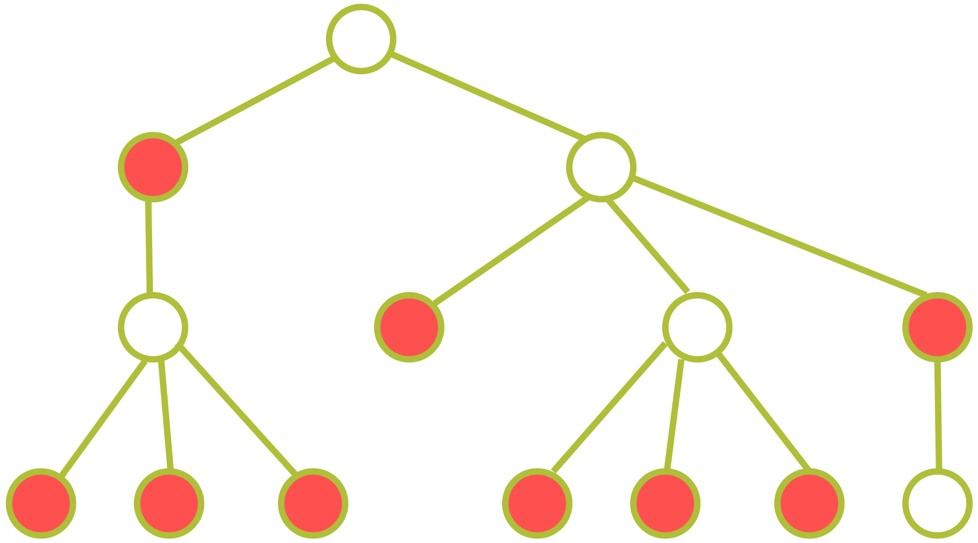
```
need(X) :- input(X).     % X = original input to problem
need(X) :- need(t(R,[X|_])).
need(t(R,Xs)) :- need(t(R,[_|Xs])).
```

- “Magic sets” transformation: only build what we need. E.g.,

```
rooted(t(R,[X|Xs])) = unrooted(X) + rooted(t(R,Xs))
                            if need(t(R,[X|Xs])).
```

Okay, that should work …

- In this example, if everyone has IQ = 1,
  the maximum total IQ on the board is 9.
- So the program finds goal = 9.
- Let's use the visual debugger, Dynasty, to see a trace of its computations.



Edit distance between two strings

4 edits
```
clara
caca
```

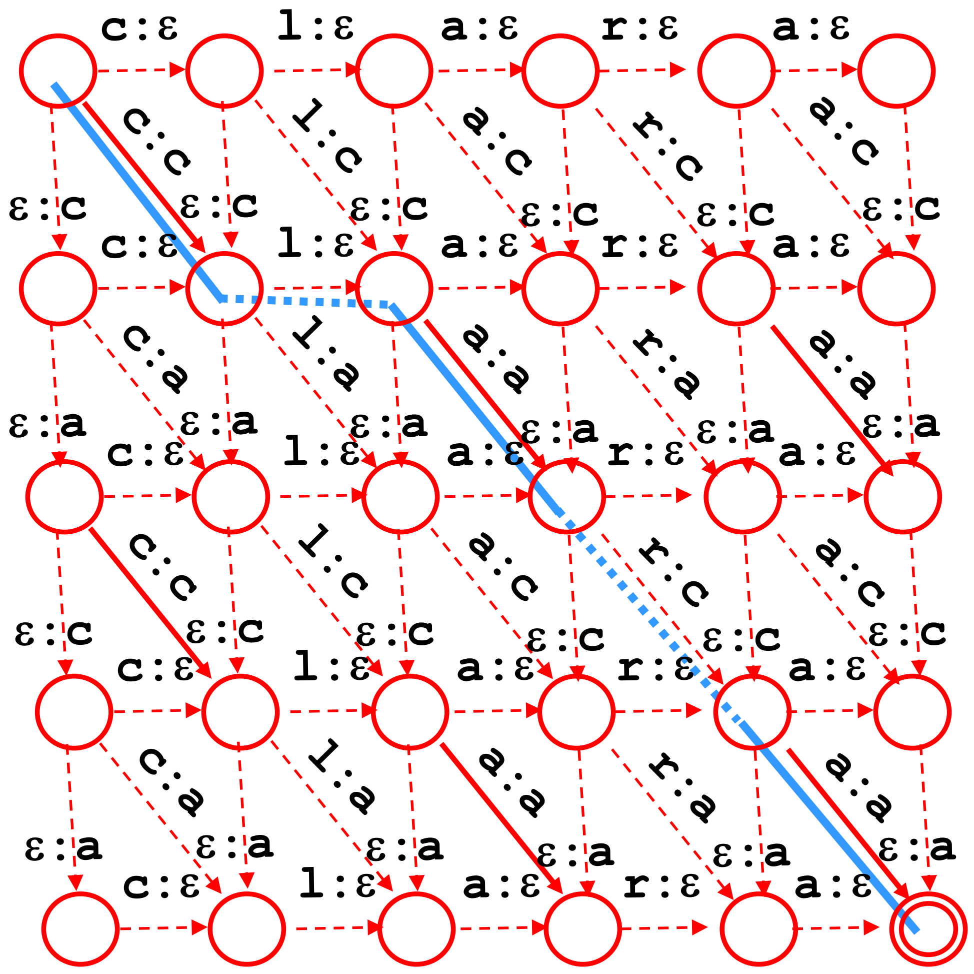
3 edits
```
clara
 caca
```

2
```
clara
c aca
```

3
```
cla ra
c ac a
```

9
```
clara
     caca
```

Traditional picture



Edit distance in Dyna: version 1

- letter1("c",0,1). letter1("l",1,2). letter1("a",2,3). … % clara
- letter2("c",0,1). letter2("a",1,2). letter2("c",2,3). … % caca
- end1(5). end2(4). delcost := 1. inscost := 1. substcost := 1.

Cost of best alignment of first I1 characters of string 1 with first I2 characters of string 2.

- align(0,0) min= 0.

Next letter is L2. Add it to string 2 only.

- align(I1,J2) min= $\boxed{\text{align(I1,I2)}}$ + $\boxed{\text{letter2(L2,I2,J2)}}$ + $\boxed{\text{inscost(L2).}}$
- align(J1,I2) min= align(I1,I2) + letter1(L1,I1,J1) + delcost(L1).
- align(J1,J2) min= align(I1,I2) + letter1(L1,I1,J1) + letter2(L2,I2,J2) + subcost(L1,L2).
- align(J1,J2) min= align(I1,I2)+letter1(L,I1,J1)+letter2(L,I2,J2).

same L; free move!

- goal min= align(N1,N2) whenever end1(N1) & end2(N2).

Edit distance in Dyna: version 2

- input([“c”, “l”, “a”, “r”, “a”], [“c”, “a”, “c”, “a”]) := 0.
- delcost := 1. inscost := 1. substcost := 1.

Xs and Ys are still-unaligned suffixes.
This item's value is supposed to be cost of
aligning everything **up to but not including** them.

- alignupto(Xs,Ys) min= input(Xs,Ys).
- alignupto(Xs,Ys) min= alignupto([X|Xs],Ys) + delcost.
- alignupto(Xs,Ys) min= alignupto(Xs,[Y|Ys]) + inscost.
- alignupto(Xs,Ys) min= alignupto([X|Xs],[Y|Ys])+substcost.
- alignupto(Xs,Ys) min= alignupto([L|Xs],[L|Ys]).
- goal min= alignupto([], []).

How about different costs
for different letters?

Edit distance in Dyna: version 2

- input([“c”, “l”, “a”, “r”, “a”], [“c”, “a”, “c”, “a”]) := 0.
- delcost := 1. inscost := 1. substcost := 1.

Xs and Ys are still-unaligned suffixes.
This item's value is supposed to be cost of
aligning everything **up to but not including** them.

- alignupto(Xs,Ys) min= input(Xs,Ys).
- alignupto(Xs,Ys) min= alignupto([X|Xs],Ys) + delcost(X)
- alignupto(Xs,Ys) min= alignupto(Xs,[Y|Ys]) + inscost(Y)
- alignupto(Xs,Ys) min= alignupto([X|Xs],[Y|Ys])+substcost. (X,Y)
- alignupto(Xs,Ys) min= alignupto([L|Xs],[L|Ys]).
- goal min= alignupto([], []).

+ nocost(L,L)

☺

What is the solver doing?

- Forward-chaining
- “Chart” of values known so far
    - Stores values for reuse: dynamic programming
- “Agenda” of updates not yet processed
    - No commitment to order of processing

Remember our edit distance program

- input([“c”, “l”, “a”, “r”, “a”], [“c”, “a”, “c”, “a”]) := 0.
- delcost := 1. inscost := 1. subcost := 1.

Xs and Ys are still-unaligned suffixes.
This item's value is supposed to be cost of
aligning everything **up to but not including** them.

- alignupto(Xs,Ys) min= input(Xs,Ys).
- alignupto(Xs,Ys) min= alignupto([X|Xs],Ys) + delcost(X)
- alignupto(Xs,Ys) min= alignupto(Xs,[Y|Ys]) + inscost(Y)
- alignupto(Xs,Ys) min= alignupto([X|Xs],[Y|Ys])+subcost.
  (X,Y)
- alignupto(Xs,Ys) min= alignupto([L|Xs],[L|Ys]).
- goal min= alignupto([], []).

How does forward chaining work here?

- alignupto(Xs,Ys) min= alignupto([X|Xs],Ys) + delcost(X).
- alignupto(Xs,Ys) min= alignupto(Xs,[Y|Ys]) + inscost(Y).
- alignupto(Xs,Ys) min= alignupto([X|Xs],[Y|Ys]) + subcost(X,Y).
- alignupto(Xs,Ys) min= alignupto([A|Xs],[A|Ys]).
- Would Prolog terminate on this one?
  (or rather, on a boolean version with :- instead of min=)
- No, but Dyna does.
- What does it actually have to do?
  - alignupto([“l”, “a”, “r”, “a”], [“c”, “a”]) = 1 pops off the agenda
  - Now the following changes have to go on the agenda:
    alignupto([“a”, “r”, “a”], [“c”, “a”]) min= 1+delcost(“l”)
    alignupto([“l”, “a”, “r”, “a”], [“a”]) min= 1+inscost(“c”)
    alignupto([“a”, “r”, “a”], [“a”]) min= 1+subcost(“l”,”c”)

The "build loop"

```
alignupto(Xs,Ys)  min= alignupto([X|Xs],Ys)      + delcost(X).
alignupto(Xs,Ys)  min= alignupto(Xs,[Y|Ys])      + inscost(Y).
alignupto(Xs,Ys)  min= alignupto([A|Xs],[A|Ys]).
alignupto(Xs,Ys)  min= alignupto([X|Xs],[Y|Ys])  + subcost(X,Y).
```

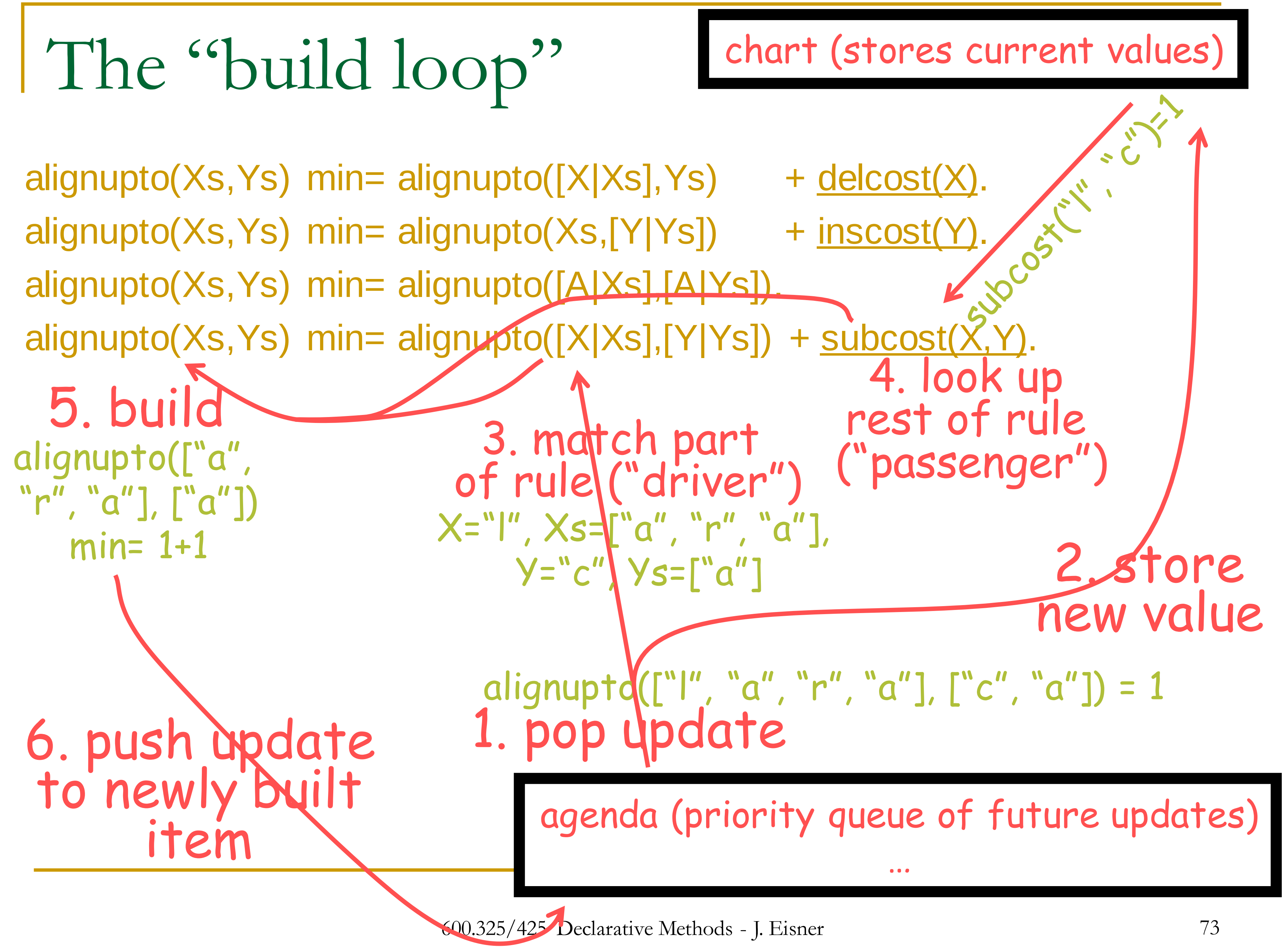


chart (stores current values)
subcost("l", "c")=1
1. pop update
alignupto(["l", "a", "r", "a"], ["c", "a"]) = 1
2. store new value
3. match part of rule ("driver")
X="l", Xs=["a", "r", "a"], Y="c", Ys=["a"]
4. look up rest of rule ("passenger")
5. build
alignupto(["a", "r", "a"], ["a"]) min= 1+1
6. push update to newly built item
agenda (priority queue of future updates)
…

The "build loop"

chart (stores current values)

```
alignupto(Xs,Ys)  min= alignupto([X|Xs],Ys)       + delcost(X).
alignupto(Xs,Ys)  min= alignupto(Xs,[Y|Ys])       + inscost(Y).
alignupto(Xs,Ys)  min= alignupto([A|Xs],[A|Ys]).
alignupto(Xs,Ys)  min= alignupto([X|Xs],[Y|Ys])  + subcost(X,Y).
```

subcost("l", "c")=1

4. look up rest of rule ("passenger")

3. match part of rule ("driver")

X="l", Xs=["a", "r", "a"],
Y="c", Ys=["a"]

2. store new value

alignupto(["l", "a", "r", "a"], ["c", "a"]) = 1

1. pop update

agenda (priority queue of future updates)

…

Might be many ways to do step 3. Why?

Dyna does all of them! Why?

Same for step 4. Why? Try this:
foo(X,Z) min= bar(X,Y) + baz(Y,Z).

The “build loop”

chart (stores current values)

```
alignupto(Xs,Ys) min= alignupto([X|Xs],Ys)      + delcost(X).
alignupto(Xs,Ys) min= alignupto(Xs,[Y|Ys])      + inscost(Y).
alignupto(Xs,Ys) min= alignupto([A|Xs],[A|Ys]).
alignupto(Xs,Ys) min= alignupto([X|Xs],[Y|Ys])  + subcost(X,Y).
```

Step 3: When an update pops, how do we quickly figure out which rules match?

Compiles to a tree of “if” tests …

Multiple matches ok.

```
if (x.root = alignupto)
   if (x.arg0.root = cons)
       matched rule 1
       if (x.arg1.root = cons)
          matched rule 4
          if (x.arg0.arg0 = x.arg1.arg0)
             matched rule 3
   if (x.arg1.root = cons)
      matched rule 2
else if (x.root = delcost)
   matched other half of rule 1
   …
```

checks whether the two A’s are equal. Can we avoid “deep equality-testing” of complex objects?

of

1

The "build loop"

chart (stores current values)

subcost("l", "c")=1

```
alignupto(Xs,Ys) min= alignupto([X|Xs],Ys)      + delcost(X).
alignupto(Xs,Ys) min= alignupto(Xs,[Y|Ys])      + inscost(Y).
alignupto(Xs,Ys) min= alignupto([A|Xs],[A|Ys]).
alignupto(Xs,Ys) min= alignupto([X|Xs],[Y|Ys])  + subcost(X,Y).
```

4. look up rest of rule ("passenger")

Step 4: For each match to a driver, how do we look up all the possible passengers?

The hard case is on the next slide …

3. match part of rule ("driver")

X="l", Xs=["a", "r", "a"],
Y="c", Ys=["a"]

alignupto(["l", "a", "r", "a"], ["c", "a"]) = 1

1. pop update

agenda (priority queue of future updates)
…

The "build loop"

chart (stores current values)

alignupto(Xs,Ys) min= alignupto([X|Xs],Ys) ...st(X).
alignupto(Xs,Ys) min= alignupto(Xs,[Y|Y ...cost(Y).
alignupto(Xs,Ys) min= alignupto([A|X
alignupto(Xs,Ys) min= alignupto([X|Xs],[Y|Ys]) + <u>subcost(X,Y)</u>.

matches <u>many</u> alignupto items in the chart!

Step 2: When adding a new item to the chart, also add it to indices so we can find it fast.

4. look up rest of rule ("passenger")

Like a Prolog query:
alignupto(["l"|Xs],["c"|Ys]).

3. match part of rule ("driver")

X="l", Y="c"

2. store new value

1. pop update

subcost("l","c") = 1

agenda (priority queue of future updates)
…

Step 4: For each match to a driver, how do we look up all the possible passengers?

Now it's an update to subcost(X,Y) that popped and is driving …

There might be <u>many</u> passengers. Look up a linked list of them in an **index:** hashtable["l", "c"].

The “build loop”

chart (stores current values)

```
alignupto(Xs,Ys)  min= alignupto([X|Xs],Ys)       + delcost(X).
alignupto(Xs,Ys)  min= alignupto(Xs,[Y|Ys])       + inscost(Y).
alignupto(Xs,Ys)  min= alignupto([A|Xs],[A|Ys]).
alignupto(Xs,Ys)  min= alignupto([X|Xs],[Y|Ys])  + subcost(X,Y).
```

subcost(“l”, “c”)=1

5. build
alignupto([“a”, “r”, “a”], [“a”])
min= 1+1

3. match part of rule (“driver”)
X=“l”, Xs=[“a”, “r”, “a”],
Y=“c”, Ys=[“a”]

4. look up rest of rule (“passenger”)

Step 5: How do we build quickly?

Answer #1: Avoid deep copies of Xs and Ys. (Just copy pointers to them.)

Answer #2: For a rule like “pathto(Y) min= pathto(X) + edge(X,Y)”, need to get fast from Y to pathto(Y). Store these items next to each other, or have them point to each other. Such memory layout tricks are needed in order to match “obvious” human implementations of graphs.

The "build loop"

chart (stores current values)

```
alignupto(Xs,Ys)  min= alignupto([X|Xs],Ys)      + delcost(X).
alignupto(Xs,Ys)  min= alignupto(Xs,[Y|Ys])      + inscost(Y).
alignupto(Xs,Ys)  min= alignupto([A|Xs],[A|Ys]).
alignupto(Xs,Ys)  min= alignupto([X|Xs],[Y|Ys])  + subcost(X,Y).
```

5. build

alignupto(["a", "r", "a"], ["a"]) min= 1+1

6. push update to newly built item

Step 6: How do we push new updates quickly?

Mainly a matter of good priority queue implementation.

Another update for the same item might already be waiting on the agenda. By default, try to consolidate updates (but this costs overhead).

agenda (priority queue of future updates)

…

Game-tree analysis

All values represent total advantage to player 1 starting at this board.

- % how good is Board for player 1, if it's **player 1's move**?
- best(Board) max= stop(player1, Board).
- best(Board) max=
  move(player1, Board, NewBoard) + worst(NewBoard).

- % how good is Board for player 1, if it's **player 2's move**?
  (player 2 is trying to make player 1 lose: zero-sum game)
- worst(Board) min= stop(player2, Board).
- worst(Board) min=
  move(player2, Board, NewBoard) + best(NewBoard).

- % How good for player 1 is the starting board?
- goal = best(Board) if start(Board).

how do we implement move, stop, start?

chaining?

Partial orderings

- Suppose you are told that

  x <= q   p <= x   p <= y   y <= p   y != q

- Can you conclude that p < q?

1. We'll only bother deriving the "basic relations" A<=B, A!=B.
   All other relations between A and B follow automatically:
   - know(A<B) |= know(A<=B) & know(A!=B).
   - know(A==B) |= know(A<=B) & know(B<=A).

   - These rules will operate continuously to derive non-basic relations whenever we get basic ones.
   - For simplicity, let's avoid using > at all: just write A>B as B<A. (Could support > as another non-basic relation if we really wanted.)

Partial orderings

- Suppose you are told that

  x <= q   p <= x   a <= y   y <= a   y != b

- Can you conclude that p < q?

1. We'll only bother deriving the "basic relations" A<=B, A!=B.
2. First, derive basic relations directly from what we were told:

   - know(A<=B) |= told(A<=B).   know(A!=B) |= told(A!=B).

   - know(A<=B) |= told(A==B).   know(A<=B) |= told(A<B).
   - know(B<=A) |= told(A==B).   know(A!=B) |= told(A<B).

Partial orderings

- Suppose you are told that

  x <= q p <= x a <= y y <= a y != b

- Can you conclude that p < q?

1. We’ll only bother deriving the “basic relations” A<=B, A!=B.
2. First, derive basic relations directly from what we were told.
3. Now, derive new basic relations by combining the old ones.
   - know(A<=C) |= know(A<=B) & know(B<=C). % transitivity
   - know(A!=C) |= know(A<=B) & know(B<C).
   - know(A!=C) |= know(A<B) & know(B<=C).
   - know(A!=C) |= know(A==B) & know(B!=C).
   - know(B!=A) |= know(A!=B). % symmetry
   - contradiction |= know(A!=A). % contradiction

Partial orderings

- Suppose you are told that

  x <= q   p <= x   a <= y   y <= a   y != b

- Can you conclude that p < q?

1. We'll only bother deriving the "basic relations" A<=B, A!=B.
2. First, derive basic relations directly from what we were told.
3. Now, derive new basic relations by combining the old ones.
4. Oh yes, one more thing. This doesn't help us derive anything new, but it's true, so we are supposed to know it, even if the user has not given us any facts to derive it from.
   - know(A<=A) |= true.

Review: Arc consistency (= 2-consistency)

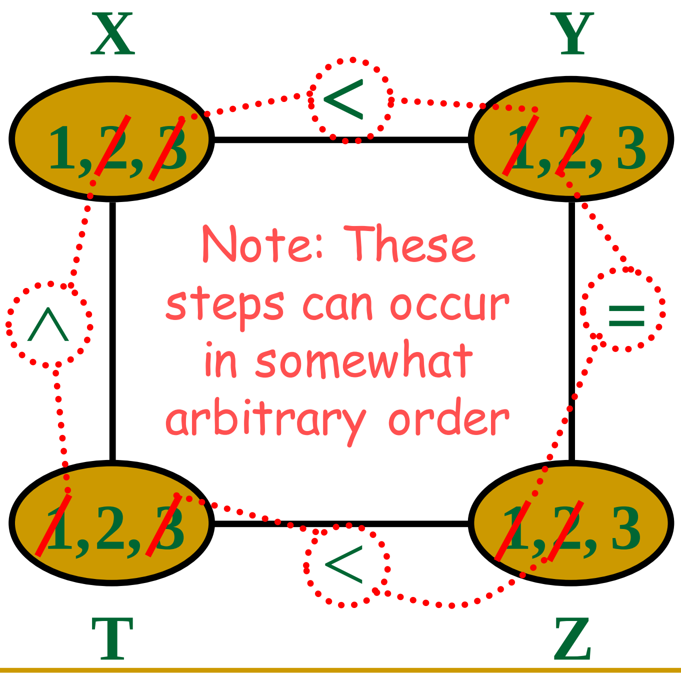
Agenda, anyone?

X:3 has no support in Y, so kill it off

Y:1 has no support in X, so kill it off

Z:1 just lost its only support in Y, so kill it off

X, Y, Z, T :: 1..3
X #< Y
Y #= Z
T #< Z
X #< T



slide thanks to Rina Dechter (modified)

Arc consistency: *The AC-4 algorithm in Dyna*

- consistent(Var:Val, Var2:Val2) := true.

  % this default can be overridden to be false for *specific* instances
  % of **consistent** (reflecting a constraint between Var and Var2)

- variable(Var) |= <u>indomain</u>(Var:Val).
- possible(Var:Val) &= <u>indomain</u>(Var:Val).
- possible(Var:Val) &= support(Var:Val, Var2)
  whenever variable(Var2).
- support(Var:Val, Var2) |= possible(Var2:Val2)
  & <u>consistent</u>(Var:Val, Var2:Val2).

Other algorithms that are nice in Dyna

- Finite-state operations (e.g., composition)
- Dynamic graph algorithms
- Every kind of parsing you can think of
  - Plus other algorithms in NLP and computational biology
  - Again, train parameters automatically
    (equivalent to inside-outside algorithm)
- Static analysis of programs
  - e.g., liveness analysis, type inference
- Theorem proving
- Simulating automata, including Turing machines

Some of our concerns

- Low-level optimizations & how to learn them
- Ordering of the agenda
  - How do you know when you've converged?
  - When does ordering affect termination?
  - When does it even affect the answer you get?
  - How could you determine it automatically?
  - Agenda ordering as a machine learning problem
  - More control strategies (backward chaining, parallelization)
- Semantics of default values
- Optimizations through program transformation
- Forgetting things to save memory and/or work: caching and pruning
- Algorithm animation & more in the debugger
- Control & extensibility from C++ side
  - new primitive types; foreign axioms; queries;
    peeking at the computation