Fun with Semirings: Dynamic Programming and Beyond

Christopher R. Genovese

Department of Statistics & Data Science

Thu 13 Nov 2025
Session #21

Plan

Something More Needed

Plan

Something More Needed

Semirings

Plan

Something More Needed

Semirings

Dynamic Programming Redux

Plan

Something More Needed

Semirings

Dynamic Programming Redux

Generic Algorithms

Announcements

- Lecture notes, two files in `Activities/dp`, reference paper in `Activities/dp`
- **Reading**:
  - Semirings and DP
    - Interlude F, Algebraic Structures Section 1
  - Databases
    - pgexercises.com
    - sqlbolt.com
    - sql-practice.com – Structured Practice
    - mystery.knightlab.com – An SQL Murder Mystery
    - selectstarql.com
  - Fun with FP
    - cube composer
- **Homework**:
  - **sym-spell** assignment due Mon 17 Nov.

Plan

Something More Needed

Semirings

Dynamic Programming Redux

Generic Algorithms

The Limitations of Exhaustive Search/Backtracking

You have several "patterns" consisting of short strings from some alphabet (e.g., lowercase letters). You have a collection of "designs" each of which is a string from the same alphabet. For each design, you want to determine whether it can be constructed by concatenating one or more patterns (repetitions allowed) and *how* it can be constructed.

For instance:

```
Patterns: r, wr, b, g, bwu, rb, gb, br

Designs:
  brwrr
  bggr
  gbbr
  rrbgbr
  ubwu
  bwurrg
  brgr
  bbrgwb
```

You decide to try exhaustive search with backtracking. Does this work? What might go wrong?

The Limitations of Exhaustive Search/Backtracking

You have several "patterns" consisting of short strings from some alphabet (e.g., lowercase letters). You have a collection of "designs" each of which is a string from the same alphabet. For each design, you want to determine whether it can be constructed by concatenating one or more patterns (repetitions allowed) and *how* it can be constructed.

For instance:

```
Patterns: r, wr, b, g, bwu, rb, gb, br

Designs:
  brwrr
  bggr
  gbbr
  rrbgbr
  ubwu
  bwurrg
  brgr
  bbrgwb
```

You decide to try exhaustive search with backtracking. Does this work? What might go wrong?

```
Patterns: c, cr, cre, cred, edu, crecc
```

The Limitations of Divide-and-Conquer

Consider this recursive calculation of Fibonacci numbers:

```
fib(n):
    if n == 0: return 0
    if n == 1: return 1
    return fib(n-1) + fib(n-2)
```

How does this perform?

The Limitations of Divide-and-Conquer

Consider this recursive calculation of Fibonacci numbers:

```
fib(n):
    if n == 0: return 0
    if n == 1: return 1
    return fib(n-1) + fib(n-2)
```

How does this perform?

```
fib(7) -> fib(6) -> fib(5), fib(4) -> fib(4), fib(3), fib(3), ...
          fib(5) -> fib(4), fib(3)
```

This can be *exponential* because we end up recomputing the same value multiple times. We are essentially visiting every node of a complete tree when we don't have to.

The Limitations of Divide-and-Conquer

Compare:

```
data Phi = Phi Natural Natural

implements Monoid Phi where
  munit = Phi 1 0
  mcombine (Phi a b) (Phi m n) = Phi (a*m + b*n) (a*n + b*(m + n))

fib n = extract (fastMonoidPow n (Phi 0 1))
  where extract (Phi _ b) = b

-- One possible implementation of fastMonoidPow
fastMonoidPow : Monoid a => Natural -> a -> a
fastMonoidPow n x
  | n == 0    = munit
  | n == 1    = x
  | otherwise = let xn2 = fastMonoidPow (n `div` 2) (x `mcombine` x)
                in
                  if n % 2 == 0 then xn2 else x `mcombine` xn2
```

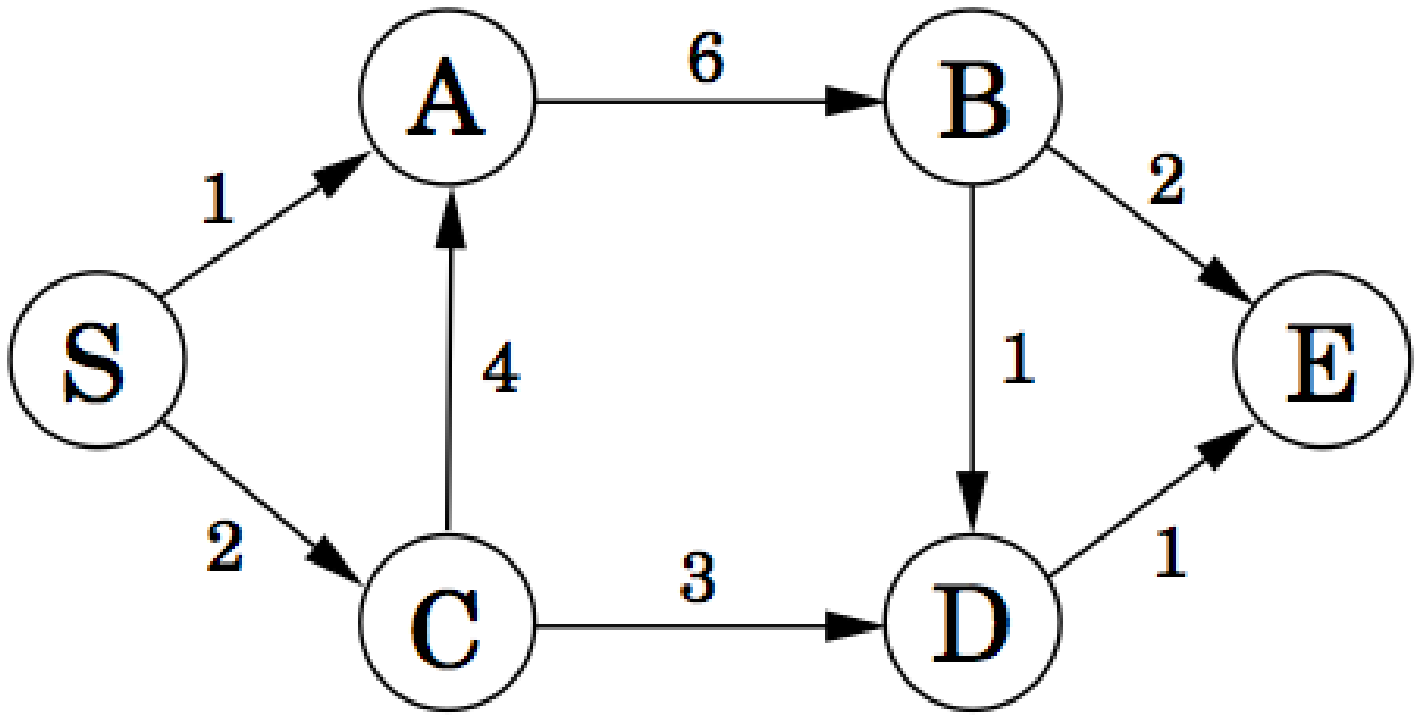
The Limitations of Divide-and-Conquer

This suggests that we consider three key strategies:

1. Find a *subproblem order* that uses our work efficiently.
   For Fib: from smaller n to larger n
   In general: arrange subproblems in a **DAG**
2. *Store the solutions* of solved subproblems to avoid redoing work.
   This is called **memoization** or caching.
3. Exploit the *algebraic structure* in how we combine subproblems.
   Many problems look identical except for a choice of **semiring**.

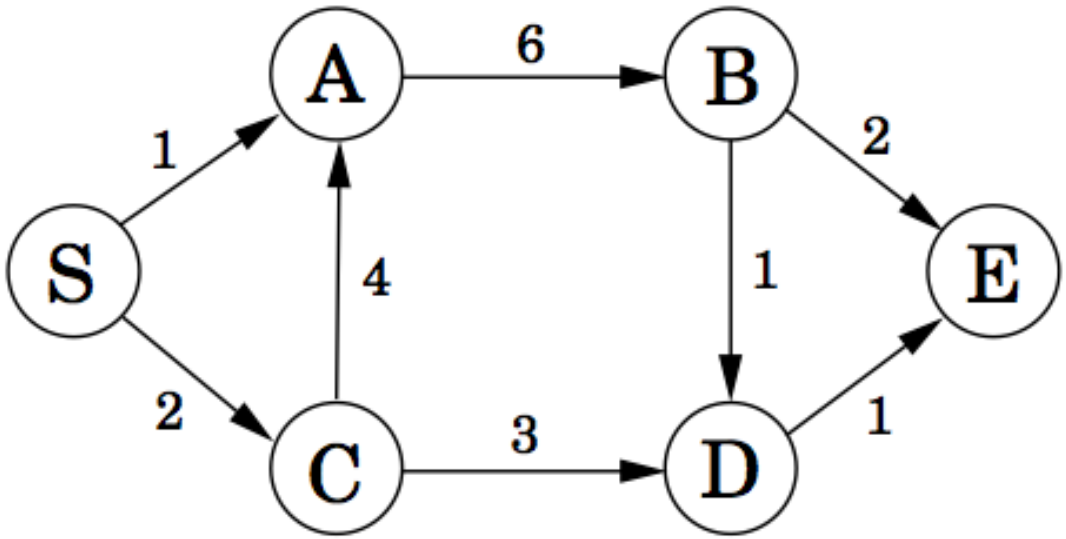
A Prototypical Problem

Consider a (one-way) road network connecting sites in a town, where each path from a site to a connected site has a cost.



What is the lowest-cost path from S to E? How do we find it? Subproblems?

A Prototypical Problem



Start from E and work backward.

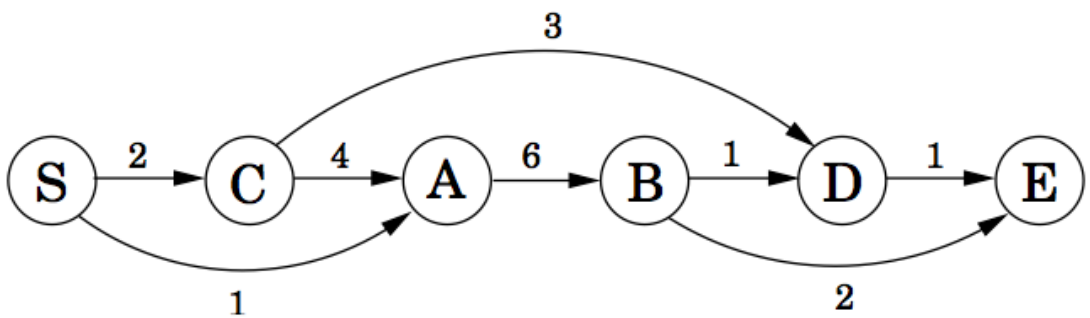
- The best route from E to E costs 0.
- The best route from D to E costs 1.
- The best route from B to E costs 2.
- The best route from A to E costs 8.
- The best route from C to E costs 4.
- The best route from S to E costs 6. [S $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E]

A Prototypical Problem (cont'd)

The lowest-cost path from a given node to E is a *subproblem* of the original.

We can arrange these subproblems in an ordering so that we can solve the easiest subproblems first and then solve the harder subproblems in terms of the easier ones.

What ordering?



This is just a ***topological sort*** of the DAG. It presents the subproblems in *decreasing order*: `S`, `C`, `A`, `B`, `D`, `E`, though in practice we often handle the subproblems in *increasing order*.

Dynamic Programming

As with Exhaustive Search, Greedy Search, and Divide-and-Conquer, we will decompose our problem into (possibly many) ***subproblems***.

We will solve trivial subproblems directly.

Otherwise, we will solve each subproblem by combining the solutions to other subproblems.

The trick to **dynamic programming** is how we use the three insights mentioned earlier: arranging problems, memoizing, and exploiting algebraic structure.

Dynamic Programming

Basic Strategy:

1. Arrange the subproblems in the order they will be needed.
2. Compute solutions to the subproblems in order, *storing* the solution to each subproblem for later use.
3. The solution to a subproblem *combines* the solutions to earlier subproblems in a specific way.

Dynamic Programming

Detailed Strategy:

1. Arrange the subproblems in a **topologically sorted DAG**.

   Each subproblem is a node in the graph, and there is a directed edge in that graph when the result of one subproblem is required in order to solve the other.

   Topological sorting gives us subproblems in the order we need. We will call the ordering *decreasing* if edges points from superproblems to subproblems and *increasing* if the reverse. We usually **solve the subproblems in increasing order**.

2. Store the solution of each subproblem for later use if needed.

   This is called **memoization** or **caching**.

   In the increasing subproblem order, we will have all the subproblem solutions available that we need to solve the current subproblem.

3. Solve each non-trivial subproblem by *combining* the solutions to earlier subproblems through a specific mathematical relation.

   The mathematical relationship between a subproblem solution and the solution of previous subproblems is often embodied in an equation, or set of equations, called the **Bellman equations**.

   These can generally be expressed in terms of **semiring** operations.

Quick Interactive Exercise: Memoizing a Function

For the Fibonacci example, what are the sub-problems? What is the DAG? What does memoizing look like?

Quick Interactive Exercise: Memoizing a Function

Here's a basic implementation. See `fib.py` and `fib.r` in `documents/Activities/dp`.

```
function fib(n):
    r = array(n)

    r[1] = 0
    r[2] = 1

    for i in 3:n:
        r[i] = r[i - 1] + r[i - 2]

    return r[n]
```

To *memoize this function*, we create a new function that calls the original when needed and consults a data store when the arguments have been given before. This exploits the *lexical scoping* in Python and R.

The new function typically replaces the old, i.e., has the original name, e.g., `fib = memoize(fib)`.

Try it!

Quick Interactive Exercise: Memoizing a Function

You have code for a function a function `f`. How would you memoize it?

```
function memoize(f):
    memoizing_table = hash_table()

    function f_prime(...):
        arglist = list(...)
        entry = memoizing_table.lookup(arglist)

        if entry:
            return entry
        else:
            value = f(...)
            memoizing_table.insert(arglist, value)
            return value

    setattr(f_prime, 'original', f)
    return f_prime

fib = memoize(fib)
fib(50)
```

Formalizing the Shortest Path

For nodes u in our graph, let $\mathrm{dist}(u)$ be the minimal cost of a path from u to E (the end node). We want $\mathrm{dist}(S)$. Finding $\mathrm{dist}(u)$ is a **subproblem**.

For each edge $u \to v$, let $c(u, v) \equiv c(v, u)$ be the travel cost.

Algorithm:

- Initialize $\mathrm{dist}(u) = \infty$ for all u.
- Topologically sort the graph in increasing order, giving a sequence from E to S.
- Set $\mathrm{dist}(E) = 0$. This is the only trivial subproblem here.
- Work through the non-trivial nodes in increasing order. For node v, set

$$\mathrm{dist}(v) = \min_{u \to v} \left(\mathrm{dist}(u) + c(u, v)\right)$$

  These last equations are called the **Bellman equations**.

Let's try it. The increasing order is E, D, B, A, C, S, yielding:

```
dist(E) =                                      0
dist(D) = dist(E) + 1                        = 1
dist(B) = min(dist(E) + 2, dist(D) + 1)      = 2
dist(A) = dist(B) + 6                        = 8
dist(C) = min(dist(A) + 4, dist(D) + 3)      = 4
dist(S) = min(dist(A) + 1, dist(C) + 2)      = 6
```

Interactive Exercise

Write a function `min_cost_path` that returns the minimal cost path to a target node from every other node in a weighted, directed graph, along with the minimal cost. If there is no directed path from a node to the target node, the path should be empty and the cost should be infinite.

Your function should take a representation of the graph and a list of nodes in subproblem order. You can represent the graph anyway you prefer; however, one convenient interface, especially for R users, would be:

```
min_cost_path(target_node, dag_nodes, costs)
```

where `target_node` names the target node, `dag_nodes` lists all the nodes in increasing order, and costs is a *symmetric* matrix of edge weights with rows and columns arranged in consistent order. Assume: `costs[u,v]` = ∞ if there is no edge between u and v.

Note: You can use the above example as a test case.

Interactive Exercise: an implementation

Utility function:

```
constantly <- function(x) {
    return( function(z) { return(x) } )
}
```

(What does this do?)

Interactive Exercise: an implementation

```
min_cost_path <- function(target_node, dag_nodes_incr, costs) {
  target_index <- match(target_node, dag_nodes_incr)
  if ( is.na(target_index) ) stop("Target node not found")
  node_count   <- length(dag_nodes_incr)
  paths        <- setNames(vector("list", node_count), dag_nodes_incr)

  dists        <- lapply(paths, constantly(Inf))
  dists[[target_node]] <- 0
  paths[[target_node]] <- c(target_node)

  for ( node_index in (target_index+1):node_count ) {
    flows_from <- target_index:(node_index-1) # indices in *increasing* order

    step_cost <- unlist(dists[flows_from]) + costs[flows_from, node_index]
    best_step <- which.min(step_cost)
    min_dist  <- step_cost[best_step]
    if ( min_dist < Inf ) {
      dists[[node_index]] <- min_dist
      paths[[node_index]] <- c(dag_nodes_incr[node_index],
                               paths[[target_index + best_step - 1]])
    }
  }
  # Note: Previous loop would be more efficient w/better graph representation
  return( list(dists = dists, paths = paths,
               target = target_node, nodes = dag_nodes_incr, weights = costs) )
}
```

Interactive Exercise: an implementation

```
pnodes <- c("E", "D", "B", "A", "C", "S")
pcost <- matrix(c(0,   1,   2,   Inf, Inf, Inf,
                  1,   0,   1,   Inf, 3,   Inf,
                  2,   1,   0,   6,   Inf, Inf,
                  Inf, Inf, 6,   0,   4,   1,
                  Inf, 3,   Inf, 4,   0,   2,
                  Inf, Inf, Inf, 1,   2,   0), 6, 6)
psol <- min_cost_path("E", pnodes, pcost)

psol$path$S
#=> [1] "S" "C" "D" "E"
d = psol$dists
c(d[["S"]], d[["C"]], d[["A"]], d[["B"]], d[["D"]], d[["E"]])
#    C.D.E   D.E   B.E     E     E
#=>     6     4     8     2     1     0
```

Example: Edit Distance

When you make a spelling mistake, you have usually produced a "word" that is close in some sense to your target word. What does *close* mean here?

The *edit distance* between two strings is the minimum number of edits – insertions, deletions, and character substitutions – that converts one string into another.

Example: Snowy vs. Sunny. What is the edit distance?

```
Snowy -> Snnwy -> Snny -> Sunny
```

Three changes transformed one into the other.

An equivalent but easier way to look at this is to think of it as an alignment problem. We use a _ marker to indicate inserts and deletions. Here are two possible alignments of Snowy and Sunny:

```
S _ n o w y          _ S n o w _ y
S u n n _ y          S u n _ _ n y

0 1 0 1 1 0          1 1 0 1 1 1 0
Total cost: 3        Total cost: 5
```

We can convince ourselves that the minimum cost here is 3, and that is the edit distance `edit(Snowy, Sunny)`.

Example: Edit Distance

But there are many possible alignments of two strings, and searching for the best among them would be costly.

Instead, we think about how to decompose this problem into sub-problems.

Consider computing `edit(s, t)` for two strings `s[1..m]`, and `t[1..n]`. The sub-problems should be smaller versions of the same problem that *help* us solve the bigger versions.

We can look at *prefixes* of the strings `s[1..i]` and `t[1..j]` as the sub-problems, and express its solutions in terms of smaller problems of the same form. Let `E_ij = edit(s[1..i], t[1..j])`.

When we find the best alignment of these strings, the last column in the table will be one of three forms

```
s[i]      _       s[i]
 _      t[j]      t[j]
```

Example: Edit Distance

The first case gives cost `1 + edit(s[1..i-1], t[1..j])`. The second case gives `1 + edit(s[1..i], t[1..j-1])`. The third case gives (s[i] != t[j]) + edit(s[1..i-1], t[1..j-1]) . Hence,

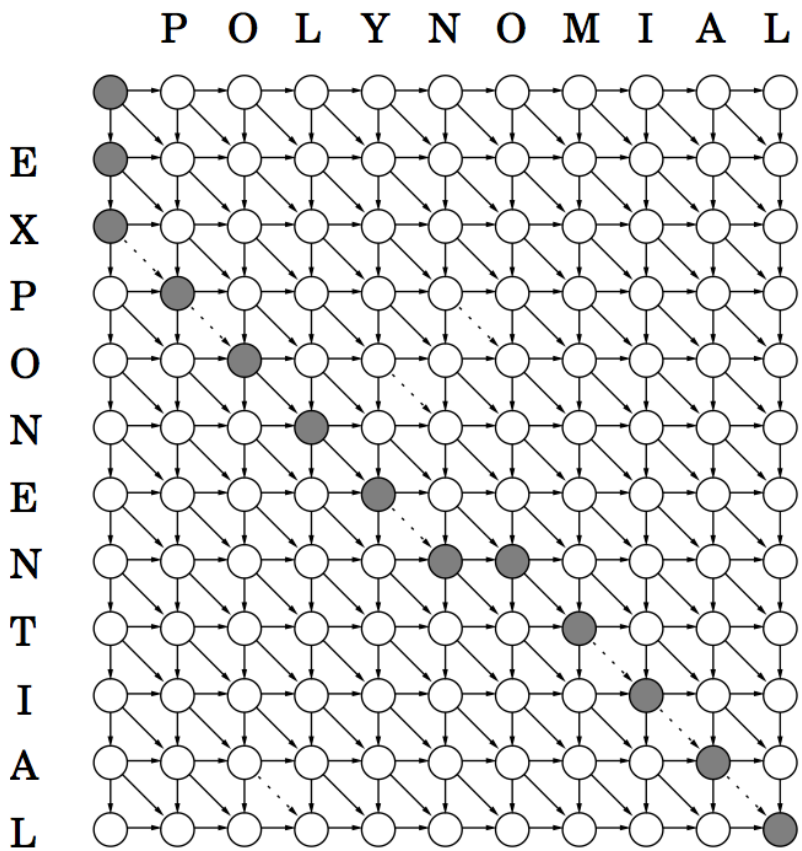
$$E_{ij} = \min(1 + E_{i-1,j}, 1 + E_{i,j-1}, d_{ij} + E_{i-1,j-1}).$$

We also have the initial conditions $E_{i0} = i$ and $E_{0j} = j$.

Hence, we can make a two dimensional table of sub-problems, which we can tackle in any order as long as E_{ij} is computed after $E_{i-1,j}, E_{i,j-1}, E_{i-1,j-1}$.

What is the DAG here? If we weight the edges, we can get generalized costs.

Example: Edit Distance



Write a function to compute the edit distance between two strings *and* a sequence of transformations of one string into the other.

What information do you need to keep track of? How do you organize the data? What types do you need?

Apply this to EXPONENTIAL and POLYNOMIAL

Plan

Something More Needed

Semirings

Dynamic Programming Redux

Generic Algorithms

Interacting Monoids: Semirings

Definition: Semirings

We say that $\langle \mathcal{S}, \boxplus, \mathbf{0}, \boxdot, \mathbf{1} \rangle$ is a **semiring** when $\mathcal{S}$ is a set with two special elements, denoted by $\mathbf{0}$ and $\mathbf{1}$, and two operators $\boxplus$ and $\boxdot$: $\mathcal{S} \times \mathcal{S} \longrightarrow \mathcal{S}$ that satisfy:

1. $\langle \mathcal{S}, \boxplus, \mathbf{0} \rangle$ is a *commutative* monoid
2. $\langle \mathcal{S}, \boxdot, \mathbf{1} \rangle$ is a monoid
3. $\mathbf{0}$ annihilates: $x \boxdot \mathbf{0} = \mathbf{0} = \mathbf{0} \boxdot x$
4. $\boxdot$ distributes over $\boxplus$:

$$a \boxdot (b \boxplus c) = (a \boxdot b) \boxplus (a \boxdot c)$$
$$(b \boxplus c) \boxdot a = (b \boxdot a) \boxplus (c \boxdot a).$$

The operator $\boxdot$ need not be commutative; if $\boxdot$ is commutative, we call this a **commutative semiring**.

In algorithmic terms, we will think of $\boxplus$ as combining different results and of $\boxdot$ as combining different choices within a result.

Interacting Monoids: Semirings (cont'd)

A **star semiring** is a semiring with an additional operation * defined by

$$a^* = 1 \boxplus (a \boxdot a^*) = 1 \boxplus (a^* \boxdot a).$$

We often are interested in *complete* star semirings, in which

$$a^* = \boxplus_{k \geq 0} a^k,$$

where $a^0 = \mathbf{1}$ and $a^k = a \boxdot a^{k-1} = a^{k-1} \boxdot a$.

We'll see how this is useful later.

Example Semirings

- Boolean logic $\langle \{\top, \bot\}, \vee, \bot, \wedge, \top \rangle$
- Natural Sum-Product $\langle \mathbb{N}, +, 0, \cdot, 1 \rangle$

  (Extends to integers and reals and complex numbers.)
- Subsets of $\mathcal{A}$ $\langle 2^{\mathcal{A}}, \cup, \{\}, \cap, \mathcal{A} \rangle$
- Square matrices with matrix sum and product
- Polynomials (and sequences) with coefficients in a semiring, with sum and product.
- Relations $\langle 2^{\mathcal{S} \times \mathcal{S}}, \cup, \{\}, \circ, \Delta \rangle$ where $\Delta = \{\langle s, s \rangle$ such that $s \in \mathcal{S}\}$ is the "diagonal" relation.
- *union*-$\times$ Semiring (up to isomorphism)

  What are the identity elements?
- Regular Languages
- ...

Example Semirings: Numeric Semirings

- *Min-Plus*. The set $\mathbb{R} \cup \{\infty\}$,is a commutative semiring with $\boxplus = \min$, $\boxdot = +$, $\mathbf{0} = \infty$, and $\mathbf{1} = 0$.

- *Max-Plus*. The set $\mathbb{R} \cup \{-\infty\}$,is a commutative semiring with $\boxplus = \min$, $\boxdot = +$, $\mathbf{0} = -\infty$, and $\mathbf{1} = 0$.

- *Max-Min*. The set $[-\infty\_\infty]$, which includes $\pm\infty$, is a commutative semiring with $\boxplus = \max$, $\boxdot = \min$, $\mathbf{0} = -\infty$, and $\mathbf{1} = \infty$.

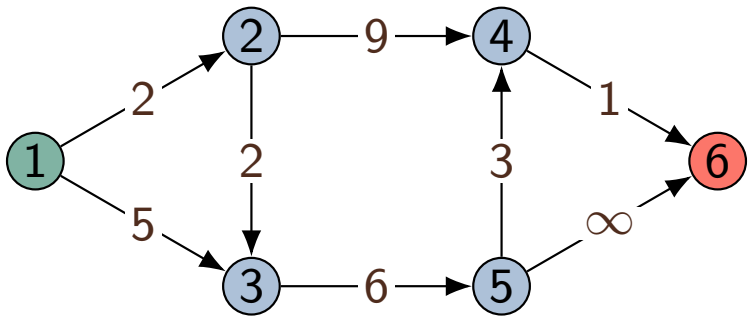
- *Max-Times*. The set $[0\_)$ is a commutative semiring with $\boxplus = \max$ the maximum of two numbers, $\boxdot = \cdot$ regular multiplication, $\mathbf{0} = 0$, and $\mathbf{1} = 1$.

- *Min-Times*. The set $(0\_\infty]$, which includes ∞, is a commutative semiring with $\boxplus = \min$, $\boxdot = \cdot$, $\mathbf{0} = \infty$, and $\mathbf{1} = 1$.

Example Semirings: Numeric Semirings

Min-Plus. The set $\mathbb{R} \cup \{\infty\}$,is a commutative semiring with $\boxplus = \min$, $\boxdot = +$, $\mathbf{0} = \infty$, and $\mathbf{1} = 0$.

Think of these as distances or costs in moving from source to target.



Look at every path from node 1 to node 6. Along each path, we combine distances with $\boxdot$ and *across* paths we aggregate with $\boxplus$. This gives us

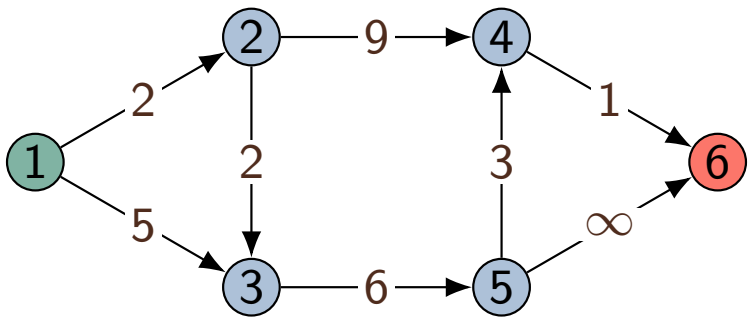
$$\begin{aligned}(2 \boxdot 9 \boxdot 1) \boxplus (2 \boxdot 2 \boxdot 6 \boxdot 3 \boxdot 1) \boxplus (2 \boxdot 2 \boxdot 6 \boxdot \infty) \\ \boxplus (2 \boxdot 5 \boxdot 6 \boxdot 3 \boxdot 1) \boxplus (2 \boxdot 5 \boxdot 6 \boxdot \infty) \\ = \min(12, 14, \infty, 17, \infty) = 12.\end{aligned}$$

The Min-Plus semiring operations gives us the minimum distance/cost over all paths from node 1 to node 6.

Example Semirings: Numeric Semirings

Max-Min. The set $[-\infty \_ \infty]$, which includes $\pm\infty$, is a commutative semiring with $\boxplus = \max$, $\boxdot = \min$, $\mathbf{0} = -\infty$, and $\mathbf{1} = \infty$.

Now, edge weights represent the capacity of flow on a channel along that edge.



We combine capacities along the paths from 1 to 6 using the semiring operations:

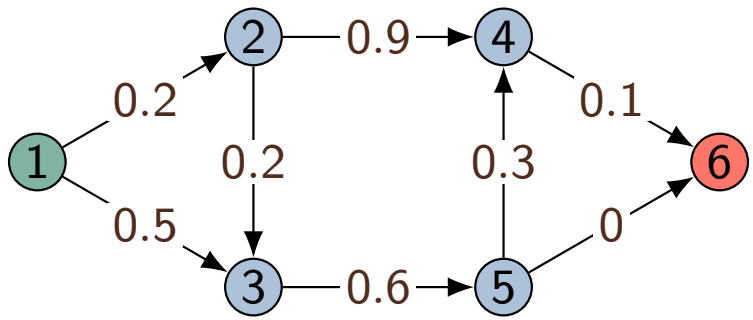
$$\begin{aligned}(2 \boxdot 9 \boxdot 1) \boxplus (2 \boxdot 2 \boxdot 6 \boxdot 3 \boxdot 1) \boxplus (2 \boxdot 2 \boxdot 6 \boxdot \infty) \\ \boxplus (2 \boxdot 5 \boxdot 6 \boxdot 3 \boxdot 1) \boxplus (2 \boxdot 5 \boxdot 6 \boxdot \infty) \\ = \max(1, 1, 2, 1, 2) = 2.\end{aligned}$$

The Max-Min semiring operations gives us optimal capacity of flow from nodes 1 to 6.

Example Semirings: Numeric Semirings

Max-Times. The set $[0\_)$ is a commutative semiring with $\boxplus = \max$, $\boxdot = \cdot$, $\mathbf{0} = 0$, and $\mathbf{1} = 1$.

Decorate the edges with the *reliability* of the connection represented by the edge. Here, reliability is measured by a number between 0 and 1 (i.e., a probability). We use a variant of the previous graph.



Aggregating again over paths from node 1 to node 6, we get

$$\begin{aligned}(0.2 \boxdot 0.9 \boxdot 0.1) \boxplus (0.2 \boxdot 0.2 \boxdot 0.6 \boxdot 0.3 \boxdot 0.1) \boxplus (0.2 \boxdot 0.2 \boxdot 0.6 \boxdot 0) \\ \boxplus (0.2 \boxdot 0.5 \boxdot 0.6 \boxdot 0.3 \boxdot 0.1) \boxplus (0.2 \boxdot 0.5 \boxdot 0.6 \boxdot 0) \\ = \max(0.018, 0.00072, 0, 0.0018, 0) = 0.018.\end{aligned}$$

We get the reliability of the most reliable path from node 1 to 6.

Plan

Something More Needed

Semirings

Dynamic Programming Redux

Generic Algorithms

Unifying Dynamic Programming

We will start by reconsidering dynamic programming algorithms and how we can unify them quite beautifully with a few concepts and some semirings.

The key idea is that **the Bellman equations can be expressed as applying $\boxdot$ along paths through the DAG to get a result and then combining results with $\boxplus$.**

For instance, in the edit-distance problem we have

$$E_{ij} = \min(1 + E_{i-1,j}, 1 + E_{i,j-1}, d_{ij} + E_{i-1,j-1}),$$

which should be viewed in the Min-Plus semiring.

Unifying Dynamic Programming

We will start by reconsidering dynamic programming algorithms and how we can unify them quite beautifully with a few concepts and some semirings.

We can define Semirings as a trait as we did for Monoids.

```
trait Semiring s where
  zero : s
  one  : s

  plus : s -> s -> s
  times : s -> s -> s

  (<+>) : s -> s -> s
  (<+>) = plus
  infix_left 60 <+>

  (<.>) : s -> s -> s
  (<.>) = times
  infix_left 70 <.>
```

We will assume several instances with this trait below.

Unifying Dynamic Programming

We will start by reconsidering dynamic programming algorithms and how we can unify them quite beautifully with a few concepts and some semirings.

We'll start by expressing a general dynamic programming problem and solver in TL1. (Based on Llorens and Vilar 2019.)

A problem has three parts:

```
data DPProblem problem score
  = record DPProblem where
      initial     : problem
      is_trivial  : problem -> Bool
      subproblems : problem -> List (score, problem)
```

Unifying DP Example: Edit distance

```
type EDistProblem = (String, String)    -- The words being compared
type Distance = Int

edp : EDistProblem -> DPProblem EDistProblem (MinPlus Distance)
edp words = DPProblem words is_trivial subprobs
  where
    is_trivial : EDistProblem -> Bool
    is_trivial words = words == ("", "")

    subprobs : EDistProblem -> List (Distance, EDistProblem)
    subprobs (a :: as, "") = [ (1, (as, "")) ]
    subprobs ("", b :: bs) = [ (1, ("", bs)) ]
    subprobs (a :: as, b :: bs) = [ (1, (a :: as, bs))
                                  , (1, (as, b :: bs))
                                  , (if a == b then 0 else 1, (as, bs))
                                  ]
```

Unifying DP Example: Knapsack Problem

```
type Capacity = Int
type Value    = Int
type Weight   = Int
data Item = record Item { value : Value, weight : Weight }
data KnapsackProblem = record KnP where
                         capacity : Capacity
                         items    : List Item

knp : KnapsackProblem -> DPProblem KnapsackProblem (MaxPlus Value)
knp sack = DPProblem sack is_trivial subprobs
  where
    is_trivial : KnapsackProblem -> Bool
    is_trivial (KnP _ items) = empty items

    subprobs : KnapsackProblem -> List (Value, KnapsackProblem)
    subprobs (KnP cap (Cons item rest)) =
      | item.weight <= cap = [ (0, KnP cap rest)
                             , (item.value, KnP (cap - item.weight) rest)
                             ]
      | otherwise          = [ (0, KnP cap rest) ]
```

Unifying DP Example: Dirichlet Problem

Here, we do a 1-dimensional version: what is probability that symmetric random walk reaches a state $\geq s$ in t or fewer steps? This extends easily to more general Dirichlet problems.

```
type Probability = Real
type Position = Int
type Step = Int
data DirichletProblem = record DirP where
                          start : Position
                          final : Position
                          limit : Steps

dirp : DirichletProblem -> DPProblem DirichletProblem Probability
dirp state = DPProblem state is_trivial subprobs
  where
    is_trivial : DirichletProblem -> Bool
    is_trivial d = d.start >= d.final

    subprobs : DirichletProblem -> List (Probability, DirichletProblem)
    subprobs d =
      | d.limit == 0 = []
      | otherwise    = let d_up = d { start=d.start + 1, limit=d.limit - 1 }
                           d_dn = d { start=d.start - 1, limit=d.limit - 1 }
                       in
                         [ (0.5, d_up), (0.5, d_dn) ]
```

Unified Solution to DPs

We can solve these problems with the ***same simple code***:

```
solveDP : Semiring score => DPProblem problem score -> score
solveDP dpp = go (initial dpp)
  where
    go : problem -> score
    go p
      | dpp.is_trivial p = one
      | otherwise        =
          let next = [sc <.> go subp | (sc, subp) <- dpp.subproblems p]
          in
            fold (<+>) zero next

editDistance : EDistProblem -> MinPlus Distance
editDistance = solveDP . edp

knapsack : KnapsackProblem -> MinPlus Distance
knapsack = solveDP . knp

dirichlet : DirichletProblem -> Probability
dirichlet = solveDP . dirp
```

Unified Solution to DPs (cont'd)

One issue: we have not yet *memoized*.

The solution is easy: make one change in `solveDP`:

```
solveDP : Semiring score => DPProblem problem score -> score
solveDP dpp = mem_go (initial dpp)
  where
    mem_go : problem -> score
    mem_go = memo go

    go : problem -> score
    go p
      ...
```

Getting the Solution Paths

In practice, for DP problems, we want to find not just the best score but the best solution (or all the best solutions or all solutions or how many solutions ...).

We can do this by constructing new semirings to capture what we want. For instance, for the best solution this looks like

```
data BestSolution decisions score =
  BestSolution (List (Maybe decisions)) score

implements Semiring (BestSolution ds sc) where
  BestSolution ds1 sc1 <+> BestSolution ds2 sc2
    | sc1 == optimum sc1 sc2 = BestSolution ds1 sc1
    | otherwise              = BestSolution ds2 sc2

  BestSolution ds1 sc1 <.> BestSolution ds2 sc2 =
    BestSolution (ap concat ds1 ds2) (sc1 <.> sc2)
```

With some small changes to our original code, we now get the best solution with the best score.

With tweaks to the semiring, we can get All Best Solutions, All Solutions, Count of Solutions, and more.

Plan

Something More Needed

Semirings

Dynamic Programming Redux

Generic Algorithms

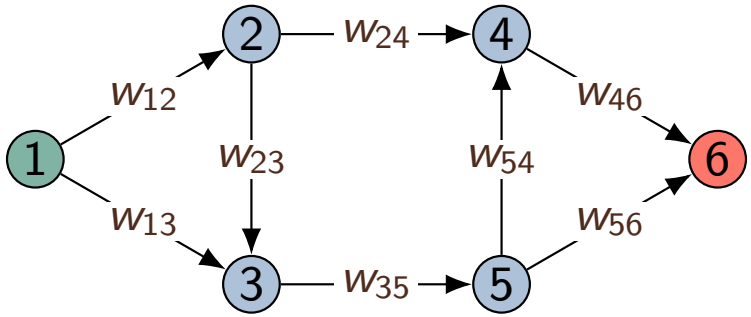
Different Problems, Same Algorithm

The generality we get from Dynamic Programming extends to a much wider class of problems.

Such problems can be solved with generic algorithms – literally ***the same code*** simply by **interpreting the objects in an appropriate semiring**.

That is, we need to define `zero`, `one`, `<.>`, and `<+>` for the entities in the problem.

Example: Path Problems



- Shortest paths: edge weights are distances, Min-Plus Semiring
- Connectivity: edge weights are Booleans, Boolean Logic Semiring
- Capacity: edge weights are capacities, Max-Min Semiring
- Reliability: edge weights are probabilities, Max-Plus Semiring
- Language Accepted: edge labels are transitions, $\cup$ −concat Semiring
- ...

All of these problems use the **exact same code**.

Path Problems (cont'd)

General structure

- We have a matrix of edge weights W
- Compute
$$P = \boxplus_{k \geq 0} W^k = I \boxplus W \boxplus (W \boxdot W) \boxplus \cdots .$$
- If this limit exists, it solves a ***fixed-point equation***
$$X = WX + I.$$

  One algorithm to rule them all. Same implementation, same complexity.

Other Problems Where This Works Include

- Satisfiability Problems

  Example: Map coloring, See Knuth vol 4, more later if time allows

- Database Queries and Joins

  Example: See (Olteanu 2022)

- Bayesian Networks

  Factor probabilities into lower dimensional marginalized terms. Same implementation with Max-Times instead of Plus-Times gives max posterior.

- ...

The End