Dynamic Programming and Equation Solving

- *The crux of a dynamic programming solution:* set up equation to captures a problem's optimal substructure.

Dynamic Programming and Equation Solving

- *The crux of a dynamic programming solution:* set up equation to captures a problem's optimal substructure.

  The equation implies dependencies on subproblem solutions.

Dynamic Programming and Equation Solving

- *The crux of a dynamic programming solution:* set up equation to captures a problem's optimal substructure.

  The equation implies dependencies on subproblem solutions.

- *Dynamic programming algorithm:* finds a schedule that respects these dependencies

Dynamic Programming and Equation Solving

- *The crux of a dynamic programming solution:* set up equation to captures a problem's optimal substructure.

  The equation implies dependencies on subproblem solutions.
- *Dynamic programming algorithm:* finds a schedule that respects these dependencies
- *Typically, dependencies form a DAG:* its topological sort yields the right schedule

Dynamic Programming and Equation Solving

- *The crux of a dynamic programming solution:* set up equation to captures a problem's optimal substructure.

  The equation implies dependencies on subproblem solutions.
- *Dynamic programming algorithm:* finds a schedule that respects these dependencies
- *Typically, dependencies form a DAG:* its topological sort yields the right schedule
- *Cyclic dependencies:* What if dependencies don't form a DAG, but is a general graph.

Dynamic Programming and Equation Solving

- *The crux of a dynamic programming solution:* set up equation to captures a problem's optimal substructure.

  The equation implies dependencies on subproblem solutions.
- *Dynamic programming algorithm:* finds a schedule that respects these dependencies
- *Typically, dependencies form a DAG:* its topological sort yields the right schedule
- *Cyclic dependencies:* What if dependencies don't form a DAG, but is a general graph.
- *Key Idea:* Use iterative techniques to solve (recursive) equations

Fixpoints

- A fixpoint is a solution to an equation:
- Substitute the solution on the rhs, it yields the lhs.

Fixpoints

- A fixpoint is a solution to an equation:
- Substitute the solution on the rhs, it yields the lhs.
- *Example 1:* $y = y^2 - 12$.
  - A fixpoint is $y = 4$:
    $$y = y^2 - 12\Big|_{y=4} = 4^2 - 12 = 4$$
    i.e., substituting $y = 4$ on the rhs returns the same value for y.
  - A second fix point is $y = -3$

Fixpoints (2)

- A fixpoint is a solution to an equation:
  - *Example 2:* $7x = 2y - 4, 2xy = 2x^3 + 2y + x$.
    - First, rewrite it to expose the fixpoint structure better:

$$x = (2y - 4)/7, \quad y = x^2 + y/x + 0.5$$

One fixpoint is $x = 2, y = 9$.

$$x = (2y - 4)/7 \,\Big|_{x=2,y=9} = (18 - 4)/7 = 2$$

$$y = x^2 + y/x + 0.5 \,\Big|_{x=2,y=9} = 2^2 + 9/2 + 0.5 = 9$$

Again, we get the same values after substitution, i.e., a fixpoint.

Fixpoints (2)

- A fixpoint is a solution to an equation:
  - *Example 2:* $7x = 2y - 4, 2xy = 2x^3 + 2y + x$.
    - First, rewrite it to expose the fixpoint structure better:

$$x = (2y - 4)/7, \quad y = x^2 + y/x + 0.5$$

One fixpoint is $x = 2, y = 9$.

$$x = (2y - 4)/7 \,\Big|_{x=2,y=9} = (18 - 4)/7 = 2$$

$$y = x^2 + y/x + 0.5 \,\Big|_{x=2,y=9} = 2^2 + 9/2 + 0.5 = 9$$

Again, we get the same values after substitution, i.e., a fixpoint.

- The term "fixpoint" emphasizes an iterative strategy.

Fixpoints (2)

- A fixpoint is a solution to an equation:
  - *Example 2:* $7x = 2y - 4, 2xy = 2x^3 + 2y + x$.
    - First, rewrite it to expose the fixpoint structure better:

$$x = (2y-4)/7, \quad y = x^2 + y/x + 0.5$$

One fixpoint is $x = 2, y = 9$.

$$x = (2y-4)/7 \,\Big|_{x=2,y=9} = (18-4)/7 = 2$$

$$y = x^2 + y/x + 0.5 \,\Big|_{x=2,y=9} = 2^2 + 9/2 + 0.5 = 9$$

Again, we get the same values after substitution, i.e., a fixpoint.

- The term "fixpoint" emphasizes an iterative strategy.
- *Example techniques:* Gauss-Seidel method (linear system of equations), Newton's method (finding roots), ...

Convergence

- Convergence is a major concern in iterative methods
  - *For real-values variables,* need to start close enough to the solution, or else the iterative procedure may not converge.

Convergence

- Convergence is a major concern in iterative methods
  - *For real-values variables,* need to start close enough to the solution, or else the iterative procedure may not converge.
  - *In discrete domains,* rely on *monotonicity* and *well-foundedness.*

Convergence

- Convergence is a major concern in iterative methods
  - *For real-values variables,* need to start close enough to the solution, or else the iterative procedure may not converge.
  - *In discrete domains,* rely on *monotonicity* and *well-foundedness.*

  Well-founded order: An order that has no infinite ascending chain (i.e., sequence of elements $a_0 < a_1 < a_2 < \cdots$ where there is no maximum)

Convergence

- Convergence is a major concern in iterative methods
  - *For real-values variables,* need to start close enough to the solution, or else the iterative procedure may not converge.
  - *In discrete domains,* rely on *monotonicity* and *well-foundedness.*

  Well-founded order: An order that has no infinite ascending chain (i.e., sequence of elements $a_0 < a_1 < a_2 < \cdots$ where there is no maximum)

  Monotonicity: Successive iterations produce larger values with respect to the order, i.e., $rhs|_{sol_i} \geq sol_i$

  Result: Start with an initial guess S^0, note $S^i = rhs|_{S^{i-1}}$.

Convergence

- Convergence is a major concern in iterative methods
  - *For real-values variables,* need to start close enough to the solution, or else the iterative procedure may not converge.
  - *In discrete domains,* rely on *monotonicity* and *well-foundedness.*

  Well-founded order: An order that has no infinite ascending chain (i.e., sequence of elements $a_0 < a_1 < a_2 < \cdots$ where there is no maximum)

  Monotonicity: Successive iterations produce larger values with respect to the order, i.e., $rhs|_{sol_i} \geq sol_i$

  Result: Start with an initial guess S^0, note $S^i = rhs|_{S^{i-1}}$.
    - Due to monotonicity, $S^i \geq S^{i-1}$, and

Convergence

- Convergence is a major concern in iterative methods
  - *For real-values variables,* need to start close enough to the solution, or else the iterative procedure may not converge.
  - *In discrete domains,* rely on *monotonicity* and *well-foundedness.*

  Well-founded order: An order that has no infinite ascending chain (i.e., sequence of elements $a_0 < a_1 < a_2 < \cdots$ where there is no maximum)

  Monotonicity: Successive iterations produce larger values with respect to the order, i.e., $rhs|_{sol_i} \geq sol_i$

  Result: Start with an initial guess S^0, note $S^i = rhs|_{S^{i-1}}$.

  - Due to monotonicity, $S^i \geq S^{i-1}$, and
  - by well-foundedness, the chain $S^0, S^1, \ldots$ can't go on forever.

Convergence

- Convergence is a major concern in iterative methods
  - *For real-values variables,* need to start close enough to the solution, or else the iterative procedure may not converge.
  - *In discrete domains,* rely on *monotonicity* and *well-foundedness.*

  Well-founded order: An order that has no infinite ascending chain (i.e., sequence of elements $a_0 < a_1 < a_2 < \cdots$ where there is no maximum)

  Monotonicity: Successive iterations produce larger values with respect to the order, i.e., $rhs|_{sol_i} \geq sol_i$

  Result: Start with an initial guess S^0, note $S^i = rhs|_{S^{i-1}}$.
    - Due to monotonicity, $S^i \geq S^{i-1}$, and
    - by well-foundedness, the chain $S^0, S^1, \ldots$ can't go on forever.
    - Hence iteration must converge, i.e., $\exists k\ \forall i > k\ \ S^i = S^k$

Role of Iterative Solutions

- *Fixpoint iteration resembles an inductive construction*
  - S^0 is the base case, S^i construction from S^{i-1} is the induction step.

Role of Iterative Solutions

- *Fixpoint iteration resembles an inductive construction*
  - S^0 is the base case, S^i construction from S^{i-1} is the induction step.
- Drawback of *explicit fixpoint iteration:* hard to analyze the number of iterations, and hence the runtime complexity

Role of Iterative Solutions

- *Fixpoint iteration resembles an inductive construction*
  - S^0 is the base case, S^i construction from S^{i-1} is the induction step.
- Drawback of *explicit fixpoint iteration:* hard to analyze the number of iterations, and hence the runtime complexity
- So, algorithms tend to rely on inductive, bottom-up constructions with enough detail to reason about runtime.

Role of Iterative Solutions

- *Fixpoint iteration resembles an inductive construction*
  - S^0 is the base case, S^i construction from S^{i-1} is the induction step.
- Drawback of *explicit fixpoint iteration:* hard to analyze the number of iterations, and hence the runtime complexity
- So, algorithms tend to rely on inductive, bottom-up constructions with enough detail to reason about runtime.
- Fixpoint iteration thus serves two main purposes:

Role of Iterative Solutions

- *Fixpoint iteration resembles an inductive construction*
  - S^0 is the base case, S^i construction from S^{i-1} is the induction step.
- Drawback of *explicit fixpoint iteration:* hard to analyze the number of iterations, and hence the runtime complexity
- So, algorithms tend to rely on inductive, bottom-up constructions with enough detail to reason about runtime.
- Fixpoint iteration thus serves two main purposes:
  - When it is possible to bound its complexity in advance, e.g., non-recursive definitions

Role of Iterative Solutions

- *Fixpoint iteration resembles an inductive construction*
  - S^0 is the base case, S^i construction from S^{i-1} is the induction step.
- Drawback of *explicit fixpoint iteration:* hard to analyze the number of iterations, and hence the runtime complexity
- So, algorithms tend to rely on inductive, bottom-up constructions with enough detail to reason about runtime.
- Fixpoint iteration thus serves two main purposes:
  - When it is possible to bound its complexity in advance, e.g., non-recursive definitions
  - As an intermediate step that can be manually analyzed to uncover inductive structure explicitly.

Shortest Path Problems

Graphs with cycles: Natural example where the optimal substructure equations are recursive.

Single source: $d_v = min_{u|(u,v)\in E}\ (d_u + l_{uv})$

All pairs: $d_{uv} = min_{w|(w,v)\in E}\ (d_{uw} + l_{wv})$

or, alternatively, $d_{uv} = min_{w\in V}\ (d_{uw} + d_{wv})$

Shortest Path Problems

Graphs with cycles: Natural example where the optimal substructure equations are recursive.

Single source: $d_v = min_{u|(u,v)\in E}\ (d_u + l_{uv})$

All pairs: $d_{uv} = min_{w|(w,v)\in E}\ (d_{uw} + l_{wv})$

or, alternatively, $d_{uv} = min_{w\in V}\ (d_{uw} + d_{wv})$

Our study of shortest path algorithms is based on fixpoint formulation

- Shows how different shortest path algorithms can be derived from this perspective.
- Highlights the similarities between these algorithms, making them easier to understand/remember.

Single-source shortest paths

For the source vertex s, $d_s = 0$. For $v \neq s$, we have the following equation that captures the optimal substructure of the problem. We use the convention $l_{uu} = 0$ for all u, as it simplifies the equation:

$$d_v = min_{u|(u,v)\in E}\ (d_u + l_{uv})$$

Expressing edge lengths as a matrix, this equation becomes:

$$\begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_j \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ l_{12} & l_{22} & \cdots & l_{n2} \\ \vdots & \vdots & \vdots & \vdots \\ l_{1j} & l_{2j} & \cdots & l_{jn} \\ \vdots & \vdots & \vdots & \vdots \\ l_{1n} & l_{2n} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_j \\ \vdots \\ d_n \end{bmatrix}$$

Matches the form of linear simultaneous equations, except that point-wise multiplication and addition become the integer "+" and *min* operations respectively.

Single-source shortest paths

SSP, written as a recursive matrix equation is:

$$D = \mathbf{L}D$$

Now, solve this equation iteratively:

$$\begin{aligned} D^0 &= Z \qquad (Z \text{ is the column matrix consisting of all } \infty \text{ except } d_s = 0) \\ D^1 &= \mathbf{L}Z \\ D^2 &= \mathbf{L}D^1 = \mathbf{L}(\mathbf{L}Z) = \mathbf{L}^2 Z \end{aligned}$$

Or, more generally, $D^i = \mathbf{L}^i Z$

- $\mathbf{L}$ is the generalized adjacency matrix, with entries being edge weights (aka edge lengths) rather than booleans.
- Side note: In this domain, multiplicative identity $\mathbf{I}$ is a matrix with zeroes on the main diagonal, and ∞ in all other places.
  - So, $\mathbf{L} = \mathbf{I} + \mathbf{L}$, and hence $\mathbf{L}^* = \lim_{r \to \infty} \mathbf{L}^r$

Single-source shortest paths

- Recall the connection between paths and the entries in $\mathbf{L}^i$.
- Thus, D^i represents the shortest path using i or fewer edges!
- Unless there are cycles with negative cost in the graph, all shortest paths must have a length less than n, so:

- D^n contains all of the shortest paths from the source vertex s
- d_i^n is the shortest path length from s to the vertex i.

Computing $\mathbf{L} \times \mathbf{L}$ takes $O(n^3)$, so overall SSP cost is $O(n^4)$.

SSP: Improving Efficiency of Matrix Formulation

- Compute the product from right: $(\mathbf{L} \times (\mathbf{L} \times \cdots (\mathbf{L} \times Z) \cdots)$
  - Each multiplication involves $n \times n$ and $1 \times n$ matrix, so takes $O(n^2)$ instead of $O(n^3)$ time.
  - Overall time reduced to $O(n^3)$.
- To compute $\mathbf{L} \times d_j$, enough to consider neighbors of j, and not all n vertices

$$d_j^i = min_{k|(k,j)\in E}(d_k^{i-1} + l_{kj})$$

  - Computes each matrix multiplication in $O(|E|)$ time, so we have an overall $O(|E||V|)$ algorithm.
- *We have stumbled onto the Bellman-Ford algorithm!*

Further Optimization on Iteration

$$d_j^i = min_{k|(k,j)\in E}(d_k^{i-1} + l_{kj})$$

- *Optimization 1:* If none of the d_k's on the rhs changed in the previous iteration, then d_j^i will be the same as d_j^{i-1}, so we can skip recomputing it in this iteration.
- Can be an useful improvement in practice, but asymptotic complexity unchanged from $O(|V||E|)$

Optimizing Iteration

$$d_j^i = min_{k|(k,j)\in E}(d_k^{i-1} + l_{kj}))$$

Optimization 2: Wait to update d_j on account of d_k on the rhs *until* d_k*'s cost stabilizes*

- Avoids repeated propagation of min cost from k to j — instead propagation takes place just once per edge, i.e., $O(|E|)$ times

Optimizing Iteration

$$d_j^i = min_{k|(k,j)\in E}(d_k^{i-1} + l_{kj}))$$

Optimization 2: Wait to update d_j on account of d_k on the rhs *until* d_k*'s cost stabilizes*

- Avoids repeated propagation of min cost from k to j — instead propagation takes place just once per edge, i.e., $O(|E|)$ times
- If all weights are non-negative, we can determine when costs have stabilized for a vertex k
  - There must be at least r vertices whose shortest path from the source s uses r or fewer edges.
  - In other words, if d_k^i has the rth lowest value, then d_k^i has stabilized if $r \leq i$

Optimizing Iteration

$$d_j^i = min_{k|(k,j)\in E}(d_k^{i-1} + l_{kj}))$$

Optimization 2: Wait to update d_j on account of d_k on the rhs *until d_k's cost stabilizes*

- Avoids repeated propagation of min cost from k to j — instead propagation takes place just once per edge, i.e., $O(|E|)$ times
- If all weights are non-negative, we can determine when costs have stabilized for a vertex k
  - There must be at least r vertices whose shortest path from the source s uses r or fewer edges.
  - In other words, if d_k^i has the rth lowest value, then d_k^i has stabilized if $r \leq i$

Voila! We have Dijkstra's Algorithm!

All pairs Shortest Path (I)

$$d^i_{uv} = min_{w|(w,v)\in E}(d^{i-1}_{uw} + l_{wv})$$

- Note that d_{uv} depends on d_{uw}, but not on any d_{xy}, where $x \neq u$.

All pairs Shortest Path (I)

$$d^i_{uv} = min_{w|(w,v)\in E}(d^{i-1}_{uw} + l_{wv})$$

- Note that d_{uv} depends on d_{uw}, but not on any d_{xy}, where $x \neq u$.
- So, solutions for d_{xy} don't affect d_{uv}.

All pairs Shortest Path (I)

$$d^i_{uv} = min_{w|(w,v)\in E}(d^{i-1}_{uw} + l_{wv})$$

- Note that d_{uv} depends on d_{uw}, but not on any d_{xy}, where $x \neq u$.
- So, solutions for d_{xy} don't affect d_{uv}.
- i.e., we can solve a separate SSP, each with one of the vertices as source

All pairs Shortest Path (I)

$$d^i_{uv} = min_{w|(w,v)\in E}(d^{i-1}_{uw} + l_{wv})$$

- Note that d_{uv} depends on d_{uw}, but not on any d_{xy}, where $x \neq u$.
- So, solutions for d_{xy} don't affect d_{uv}.
- i.e., we can solve a separate SSP, each with one of the vertices as source
- i.e., we run Dijkstra's $|V|$ times, overall complexity $O(|E||V|\log|V|)$

All pairs Shortest Path (II)

$$d^i_{uv} = min_{w \in E} \left(d^{i-1}_{uw} + d^{i-1}_{wv}\right)$$

Matrix formulation:

$$\mathbf{D} = \mathbf{D} \times \mathbf{D}$$

with $\mathbf{D}^0 = \mathbf{L}$.

Iterative formulation of the above equation yields

$$\mathbf{D}^i = \mathbf{L}^{2^i}$$

We need only consider paths of length $\leq n$, so stop at $i = \log n$. Thus, overall complexity is $O(n^3 \log n)$, as each step requires $O(n^3)$ multiplication.

We have just uncovered a variant of Floyd-Warshall algorithm!

- Typically used with matrix-multiplication based formulation.

All pairs Shortest Path (II)

$$d^i_{uv} = min_{w \in E}\ (d^{i-1}_{uw} + d^{i-1}_{wv})$$

Matrix formulation:

$$\mathbf{D} = \mathbf{D} \times \mathbf{D}$$

with $\mathbf{D}^0 = \mathbf{L}$.

Iterative formulation of the above equation yields

$$\mathbf{D}^i = \mathbf{L}^{2^i}$$

We need only consider paths of length $\leq n$, so stop at $i = \log n$. Thus, overall complexity is $O(n^3 \log n)$, as each step requires $O(n^3)$ multiplication.

We have just uncovered a variant of Floyd-Warshall algorithm!

- Typically used with matrix-multiplication based formulation.

Matches ASP I complexity for dense graphs $(|E| = \Theta(|V|^2))$

Further Improving ASP II

Each step has $O(n^3)$ complexity as it considers all (u, w, v) combinations

Further Improving ASP II

Each step has $O(n^3)$ complexity as it considers all (u, w, v) combinations *Note:* Blind fixpoint iteration "breaks" recursion by limiting path length.

- Converts d_{uv} into d_{uv}^i where i is the path length
- Worked well for SSP & ASP I, not so well for ASP II

Further Improving ASP II

Each step has $O(n^3)$ complexity as it considers all (u, w, v) combinations *Note:* Blind fixpoint iteration "breaks" recursion by limiting path length.

- Converts d_{uv} into d^i_{uv} where i is the path length
- Worked well for SSP & ASP I, not so well for ASP II

Can we break cycles by limiting something else, say, vertices on the path?

Further Improving ASP II

Each step has $O(n^3)$ complexity as it considers all (u, w, v) combinations *Note:* Blind fixpoint iteration “breaks” recursion by limiting path length.

- Converts d_{uv} into d^i_{uv} where i is the path length
- Worked well for SSP & ASP I, not so well for ASP II

Can we break cycles by limiting something else, say, vertices on the path?

Floyd-Warshall: Define d^k_{uv} as the shortest path from u to v that only uses intermediate vertices 1 to k.

$$d^k_{uv} = min(d^{k-1}_{uv}, d^{k-1}_{uk} + d^{k-1}_{kv})$$

Further Improving ASP II

Each step has $O(n^3)$ complexity as it considers all (u, w, v) combinations *Note:* Blind fixpoint iteration "breaks" recursion by limiting path length.

- Converts d_{uv} into d^i_{uv} where i is the path length
- Worked well for SSP & ASP I, not so well for ASP II

Can we break cycles by limiting something else, say, vertices on the path?

Floyd-Warshall: Define d^k_{uv} as the shortest path from u to v that only uses intermediate vertices 1 to k.

$$d^k_{uv} = min(d^{k-1}_{uv}, d^{k-1}_{uk} + d^{k-1}_{kv})$$

Complexity: Need n iterations to consider $k = 1, \dots, n$ but each iteration considers only n^2 pairs, so overall runtime becomes $O(n^3)$

Summary

- A versatile, robust technique to solve optimization problems
- *Key step:* Identify *optimal substructure* in the form of an equation for optimal cost

Summary

- A versatile, robust technique to solve optimization problems
- *Key step:* Identify *optimal substructure* in the form of an equation for optimal cost
- If equations are non-recursive, then either
  - identify underlying DAG, compute costs in topological order, or,
  - write down a memoized recursive procedure
- For recursive equations, "break" recursion by introducing additional parameters.
  - A fixpoint iteration can help expose such parameters.
- Remember the choices made while computing the optimal cost, use these to construct optimal solution.