|  |  |
| --- | --- |
| Lecture 1 | 2024-09-04  Week 1, Wed |
| What is An Algorithm?  * CLRS Algorithm Definition: An algorithm is any well-defined computational procedure that takes some value(s) as input and produces some value(s) as output. * 3 aspects relevant for CS41:  1. The algorithm must always halt eventually. 2. An algorithm for solving problem X must always return what problem X asks for. 3. An algorithm does not use randomness (“random” vs. “arbitrary”).  The Hiking Problem: Your friend is on the Appalachian Trail (AT). You want to meet your friend. You don’t know the trail at all but find yourself on the trail.  **Q:** How do you find your friend? Algorithm #1: Hike North until we reach our friend or if the trail ends. Hike South if we reached the Northern terminus.  **Q:** Is this a good algorithm? **A:** No. For example, if you friend is one mile south of you at the Southern terminus, you end up walking almost twice the trail. Algorithm #2: . While we don’t friend: hike miles N, hike miles S, hike miles N (returns to the origin), . Analysis: **Case 1:** friend is miles North.  Last iteration: miles.  For non-final iterations:  For :  We travel miles.  Total distance:  This algorithm travels miles if friend is miles away. | |

|  |  |
| --- | --- |
| Lecture 2 | 2024-09-06  Week 1, Fri |
| The Hiking Problem (Continued):Algorithm #3 (Binary Exponential Backoff): . Repeat until you find friend: hike miles N, hike miles S, hike miles N (returns to the origin), . Analysis:  |  |  | | --- | --- | | **Case 1** | Friend is miles North.   * Final Iteration: miles. * Other Iterations: miles for .   **Q:** What is the final iteration?  **A:** At least such that .  **Total Distance:** |  |  |  | | --- | --- | | **Case 2** | Friend is miles South.   * Final Iteration: miles. * Other Iterations: miles.   **Q:** What is the final iteration?  **A:** At least such that .  **Total Distance:** |  4 Tasks of Algorithm Design:  * Formulate problem. * Understand underlying structure. * Design algorithm. * Analyze algorithm. * Separate design & analysis, as they are separate skills.  National Resident Matching Program (NRMP): Input: List of hospitals, list of doctors. Each hospital has a preference list of doctors. Each doctor has a preference list of hospitals.  Output: List of (hospital-doctor) matchings.  For example:   |  | | --- | | Hospitals: Abington, Brandywine, CHOP, DCMH  Doctors: Alice, Bob, Carol, Dave  Output: Abington-Carol, Brandywine-Bob, DCMH-Alice, CHOP-Dave. |  * Matching: A set of (hospital-doctor) pairs where everyone is matched at most once. A matching S is perfect is everyone is matched.   For example:   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | | |

|  |  |
| --- | --- |
| Lecture 3 | 2024-09-11  Week 2, Wed |
| The lecture on 2024-09-09 was cancelled due to instructor sickness. The NRMP Problem (Continued): **Definition:** is an instability in a matching if:   1. and are in . 2. prefers to . 3. prefers to .   **Definition:** A stable matching is a perfect matching without instabilities.  **Q:** Do stable matchings always exist? If so, can we produce stable matchings efficiently?  **A:** Yes. And yes. Algorithm #1 (Gale-Shapley Intuition): Hospitals “propose” matchings while doctors can accept/reject proposals based on their preferences.   |  | | --- | | Initialize all hospitals, doctors to be free.  While there is a hospital and a doctor that the hospital hasn’t proposed to:  Choose this hospital .  the highest-ranked doctor hasn’t yet proposed to.  If is free:  , become engaged.  Else:  If prefers to the current assignment :  becomes free.  , become engaged.  Return set of engaged pairs. |   **Q:** Is this algorithm correct? (i.e., does it return a perfect matching? If so, is it stable?) **A:** Yes.  **Q:** Is it efficient? **A:** G-S algorithm terminates after at most iterations.  **Proof:**  **Idea:** Define measure of progress, show that progress is always made, and there’s only so much progress to make.  **Examples:**   1. is the number of engaged pairs after the th iteration of loop.   .  However, sometimes (we sometimes break an engagement while making one).  Therefore, this is not a good measure of progress.   1. is the number of proposals made after the th iteration of the loop.   as there are hospitals and doctors.  .  Therefore, G-S runs in time. This is the case since each loop iteration takes time, as choosing a hospital which can make a proposal might take time. | |

|  |  |
| --- | --- |
| Lecture 4 | 2024-09-13  Week 2, Fri |
| The Gale-Shapley Algorithm (Continued):Facts:  1. Hospitals always propose to doctors in decreasing preference order. 2. Doctors become engaged the first chance they get, and they stay engaged unless when they immediately get a new engagement. 3. Doctors always accept proposes in increasing preference order.   **Claim 1:** The returned by the G-S algorithm is a matching.  **Claim 2:** The matching is perfect.  **Claim 3:** is stable. Proof of Claim 3 (by Contradiction):  |  | | --- | | Assume that has an instability: , are matchings in where prefers to and prefers to .  Note that at the end of the algorithm, matched with . Also, prefers to . By Fact 1, at some point proposes to .   * Either was free but eventually broke engagement in favor of some hospital . * Or, engaged but preferred , so become engaged to … but then eventually breaks engagement for some . * Or, engaged to some and decides to reject .   In any case, at some point makes the decision . Eventually, ends up with hospital .  By Fact 3, , so prefers to , contradicting instability. |  Efficient Algorithms: **Q:** What is an efficient algorithm?  **A:** (Textbook Definition) An algorithm is efficient if it uses a polynomial amount of resources (e.g., , , , …). However, this definition is not perfect as bubble sort, which uses a polynomial runtime, is generally considered not efficient.  **Definition:** A problem is tractable if there is an efficient algorithm for it.  **Why are inefficient algorithms bad?**   |  |  |  |  | | --- | --- | --- | --- | |  |  |  |  | | 1 | 1 | 0 | 2 | | 2 | 4 | 2 | 4 | | 4 | 16 | 8 | 16 | | 10 | 100 | ~332 | 1024 | | 100 | 10,000 | ~604 | Lots (126765060022822940149670320) | | |

|  |  |
| --- | --- |
| Lecture 5 Lecture by Lila Fontes | 2024-09-16  Week 3, Mon |
| Efficient Algorithms:  * Recall that an algorithm is efficient if it uses a polynomial amount of resources.   **Q:** What are the resources?  **A:** Time, storage space, threads, hardware, electricity. Search Space & Brute Force:  * Search Space: The set of all possible solutions to a problem is the search space. * Brute Force Search Algorithm: An algorithm that checks every possible solution in the search space.  Upper Bound, Lower Bound, and Tight Bound: **Def:** Let and be functions . is if there exist constants and such that, for all , . In this case, we say “ is big of ”, or “ is asymptotically upper-bounded by ”.    **Def:** is if there exist constants and such that, for all , . In this case, we say “ is big Omega of ”, or “ is asymptotically lower bounded by ”.    **Def:** If is and is , then is .   |  |  | | --- | --- | | **Example** | .  **Claim:** is .  **Proof:** We need to find and such that for all .  Pick and , we have:  (Note that in this step we assumed .)  Thus, we have , .  **Claim:** is .  **Proof:** Let and .  (Note that in this step we assumed , as .)  Thus, we have for all .  **Claim:** is .  **Proof:** Since is both and , the definition of is satisfied. | | |

|  |  |
| --- | --- |
| Lecture 6 Lecture by Lila Fontes | 2024-09-18  Week 3, Wed |
| **Theorem:** If is and is , then is .  **Q:** If is and is , then is ?  **A:** No.  **Counter Example:** , , . We can easily check that is and is . We now need to check that is is not .  **Proof:**  In order to show that is not , we need to show that there are no possible constants and which satisfy the definition of big . We will prove this by contradiction.  Assume that there are constants and such that for all . Note that and .  However, if , then .  This is a contradiction. Therefore, the assumption must not be true.  **Claim:** If is and is , then is ?  **Proof:**  We know for and for . Multiplying gives us for .  **Theorem:** If is , then is not necessarily .  **Counter Example:** Constant functions.  **Theorem:** If is , then is not necessarily .  **Example:** and , . .  **Claim:** is not .  **Proof:**  In order to show that is not , we need to show that there are no possible constants and which satisfy the definition of big . We will prove this by contradiction.  Assume that there are constants and such that for all . Note that and .  However, if , then .  This is a contradiction. Therefore, the assumption must not be true. | |

|  |  |
| --- | --- |
| Lecture 7 Lecture by Lila Fontes | 2024-09-20  Week 3, Fri |
| Common Runtimes:  * Polynomial. * Logarithmic. * Exponential.  Polynomial Runtime: **Def:** A polynomial is a function like where is a constant. Logarithmic: **Def:** means that is the number such that .  **Fact:** For any constants , is .  **Proof:**  so . . Taking of both sides, . Thus, is . By the same reasoning (swapping and ), is .  **Fact:** Let and be such that constant. if , then is . If , then is but is not .  **Fact:** For any , is .  **Proof:**  , . We want to show that for sufficiently large . if and only if .  Therefore, is decreasing for . .  For all , . Exponential: **Def:** for constant .  **Fact:** For all and all , is .  **Lemma:** If , is , but is not .  **Proof:** If , then , . is increasing; is decreasing. Therefore, . Common Runtime Comparison:  |  |  |  |  | | --- | --- | --- | --- | | “Efficient” |  | Constant |  | |  | Logarithmic |  | |  | Linear |  | |  |  | (e.g., merge sort) | |  | Quadratic | (e.g., bubble sort, quick sort) | |  | Cubic | (e.g., triply nested loops) | |  | Polynomial |  | |  |  | Exponential | (e.g., brute force search) | |  | Factorial |  |   **Claim:** For , , is .  **Proof:** Use the log & poly fact with : is .  From Wednesday, we know that if is , then is . Therefore, , ; is .  Using the same fact again, is , is . So is .  2 is , so is and is .  Since 6 is , we can use the fact that if is and is , then is to get is . | |

|  |  |
| --- | --- |
| Lecture 8 | 2024-09-23  Week 4, Mon |
| * High-Level Pseudocode: Abstract pseudocode that can be used to argue about the correctness of the algorithm. * Low-Level Pseudocode: Pseudocode containing abstract data types which can be used to argue about both the correctness and runtime of the algorithm.  Recap (ADT List Runtime):  |  |  |  |  | | --- | --- | --- | --- | |  | Sample Operations | Array/ArrayList | LinkedList | |  | **query(*i*):** return *i*th element |  |  | | **search(*e*):** is *e* in the list? | if sorted |  | | **first():** return the first item in the list |  |  | | Dynamic Operations | **add(*e*):** add *e* to the end of the list  *// What about add to the beginning?* | if capacity  otherwise |  | | **delete(*e*):** delete *e* from the list |  | plus **search()** |  Better Runtime for Gale-Shapely: **Algorithm:**  Initialize all hospitals, doctors as free.  While there is a free hospital and a doctor that the hospital hasn’t proposed to:  Choose this hospital .  highest ranked doctor in ’s preference list that they haven’t proposed to.  If is free:  become engaged.  Else: // is engaged.  If prefers to :  become engaged.  becomes free.  Return the set of engaged pairs.  **Inputs:**   * List of hospitals: **Hospital[1, …, *n*]**. * List of doctors: **Doctor[1, …, *n*]**. * For each hospital, the list of preferences: **HPref[*h*, *i*]** // *i*th most preferred doctor in *h*’s preference list. * For each doctor, the list of preferences: **DPref[*d*, *i*]** // *i*th most preferred hospital in *d*’s preference list.   **Tasks (We Need to Do in Each Iteration):**   1. Identify a free hospital that hasn’t proposed to all the doctors. 2. Identify the highest ranked doctor that hasn’t proposed to. 3. Check to see if is free or identify who has engaged to. 4. Compare preferences of vs. in ’s preference list.   *\* Assume all input lists are arrays.*  **Task 1 (List of Free Hospitals):**   * Start out with all the hospitals free. * Delete a hospital when it becomes engaged. * Add a hospital to the free list when the engagement is broken. * Choose a hospital *h* (that is free and hasn’t proposed to all the doctors). * Thus, for preprocessing, we add each hospital to the list and return the first hospital. * Solution: Use a **LinkedList** so that we have total work inside the loop be .   **Task 4:**   * Compare to in ’s preferences. * Solution: Maintain an array of rankings. E.g., **Rank[*d*, *h*]** // Rank of *h* in *d*’s preferences. // **Rank[*d*, *h*] = *i*** if **DPref[*d*, *i*] = *h***. | |

|  |  |
| --- | --- |
| Lecture 9 | 2024-09-25  Week 4, Wed |
| Graph Algorithms:  * A graph is an ordered pair . * is the set of vertices. * is the set of edges. * Edges can be directed or undirected. * Conventions: is the number of vertices while is the number edges.  Graph Data Structures:  * Adjacency Matrix: space; an matrix where . * Adjacency List: space; list of lists where is the list of neighbors of .  |  |  |  | | --- | --- | --- | | Graph Operations | Adjacency Matrix | Adjacency List | | **Edge Query**  ***Is***; |  |  | | **search(*e*):** is *e* in the list? |  |  |   *.* Breadth-First Search (BFS): **Algorithm:**  BFS(Vertex )  Queue  Visit()  .enqueue()  while( not empty):  .dequeue() , only happens times.  for each neighbor of :  if not visited: per iteration  Visit() per iteration  .enqueue() per iteration  Visit()  .visited true  // Do stuff.  **Total Work:** Applications:  1. S-T connectivity.   Input: , .  Output: Yes if there is a path from to . No otherwise.  Idea: Maintain previous vertex, then when you find the target vertex, follow nodes back to the source vertex.   1. Connectivity.   Input: .  Output: Are all vertices connected?   1. Find all components of a graph.     Idea: Instead of keeping track if nodes are visited, maintain and .  if is not visited yet and if is in component .  For each unvisited node : , . | |

|  |  |
| --- | --- |
| Lecture 10 | 2024-09-27  Week 4, Fri |
| Bipartite Graphs:    * **Def:** is bipartite if you can partition into two sets , such that all edges have one vertex in and one vertex in . * **Alternate Definition:** is bipartite if we can color the vertices using two colors such that each edge is biochromatic (endpoints have different colors). * is the set of vertices. * is the set of edges. * Edges can be directed or undirected. * Conventions: is the number of vertices while is the number edges.  Testing Bipartiteness:  * Input: . * Output: YES, if is bipartite. NO, if is not bipartite. * Conventions: An algorithm accepts if it outputs YES. An algorithm rejects if it outputs NO.   **Algorithm:**  TestBipartiteness() // This is not the correct algorithm!  Queue  Initialize Color[] = NULL for all vertices  Pick arbitrary :  // Visit .  Color[] = blue  .enqueue()  while( not empty):  .dequeue()  for each neighbor of :  Color[]  if( == NULL):  Color[] == opposite of Color[]  .enqueue()  Else if(Color[] == Color[]):  return NO  return YES  **Potential Issue:**   * We might not visit all nodes!     **Solution:**   * After initializing, add the following code:   for all verticies :  if Color[] == NULL:  // BFS from .  return YES if we haven’t returned NO yet. Not Bipartite Graphs:  * Example: triangles.      * **Fact:** Any odd-length cycle is not bipartite. * **Fact:** Any graph containing an odd-length cycle is not bipartite. | |

|  |  |
| --- | --- |
| Lecture 11 | 2024-09-30  Week 5, Mon |
| Bipartite Graphs (Continued):  * Imagine we have a graph. We can pick up the graph by holding onto one node, and we would get a graph that’s separated into layers, with all immediate neighbors of our node in the first layer, all the second immediate neighbors in the second layer, etc.      * **Theorem:** Let be a connected graph and be the layers of the BFS search tree:  1. If there are no edges with on the same layer, then is bipartite. 2. If there is an edge with on the same layer, then is not partite.  * **Def:** A node is the Least Common Ancestor (LCA) for nodes if:  1. and are connected. 2. is visited before . 3. Among all such nodes, is closest to .  Proof of Claim 2:  |  | | --- | | Fix edges with on the same layer. Let be the Least Common Ancestor (LCA) of .  Note that is an odd length cycle. Therefore, is not bipartite. |  Directed Graphs:  * All edges are directed.    Graph Representation:  * Adjacency Matrix: if the edge exists, otherwise. * Adjacency List: Two list with one for incoming edges and one for outgoing edges.  Search Algorithms (BFS/DFS): Q: Can we search?  A: Yes.   * **Fact:** BFS() computes the set of vertices reachable from . I.e., . * **Def:** are weakly connected if there is either a path or a path in . * **Def:**  are strongly connected if there is a path and a path in . * **Note:** The definition of weakly connected nodes uses a logical OR. All strongly connected nodes are also weakly connected. * **Def:** is a strongly connected component if are strongly connected for all and no other vertices are strongly connected to some .  Exercises:  1. Identify all SCC’s (strongly connected cycles) in the following graph:     and .   1. Given a directed graph and a start vector , design an algorithm that identifies all SCC’s containing .   Algorithm: Define but with all the edges reversed:   1. Run BFS() to get all such that there is a path in . 2. Run BFS() on to get all such that there is a path in . 3. Return the intersection of 1 and 2. | |

|  |  |
| --- | --- |
| Lecture 12 | 2024-10-02  Week 5, Wed |
| Cycles:  * The smallest undirected graph cycle has 3 vertices, while the smallest directed graph cycle has 2 vertices. * **Def:** A connected graph without any cycle is called a tree. * **Def:** A graph without any cycle but can be unconnected is a forest. * **Def:** A directed graph without cycles is called a directed, acyclic graph (or a DAG). DAGs don’t need to be connected. * DAGs are useful as they can be used to analyze and avoid deadlocks in operating systems.  Topological Ordering:  * **Def:** A topological ordering in a DAG is an ordering of vertices such that for all edges . * **Fact:** If has a topological ordering, then is a DAG. * **Theorem:** If is a DAG, then has a topological ordering.  The Topological Sorting (Topsort) Problem:  * Input: Directed acyclic graph . * Output: Topological ordering of vertices. * Claim: If is a DAG then there is with no incoming edges. * Proof (Sketch): (By Contradiction) Assume all vertices have incomding edges. Pick any vertex. Walk back along incoming edges.  Algorithm: List TopSort(DAG G) {  List InCount[1, ..., n]; // Current in degree of vertex i.  Queue InZero; // Queue of vertices safe to add to TopSort.  List tSort;  // Preprocess to build Incount[...], InZero[...].  while(InZero not empty) {  v = InZero.dequeue();  tSort.insertAtTail(v);  for(each neighbor u of v) {  InCount(u) -= 1;  if(InCount(u) == 0) {  InZero.enqueue();  }  }  }  return tSort;  } | |

|  |  |
| --- | --- |
| Lecture 13 | 2024-10-04  Week 5, Fri |
| Greedy Algorithms:  * **Def:** A decision problem is a problem where output is binary (e.g., YES/NO). * **Def:** An optimization problem is a problem where we want to maximize or minimize time, profit, # jobs, etc. * Greedy Algorithms: An algorithm is greedy if it builds up a solution one step at a time, without backtracking.  The Interval Scheduling Problem:    * Input: A list of requests. Each request has a start time and a finish time . * **Def:** Two requests are compatible if they don’t overlap in time. * Output: Largest possible set of compatible requests.  Greedy Strategy for Interval Scheduling:  * Build up requests one at a time, maintaining compatibility at all times.  1. Earliest Available Start: Pick interval to minimize . Does Not Work      1. Shortest Interval: Pick to minimize . Does Not Work      1. Earliest Available Finish: Pick to minimize . Works 2. Minimize # Conflicts: Pick to minimize the number of incompatible with . Does Not Work    Algorithm: List EarliestAvailableFinish() {  List R = set of requests;  List A = [];  while(R is not empty) {  pick i in R that minimizes f(i);  add i to A;  remove i from R;  remove all j from R that are incompatible with i;  }  return A;  } Proving EAF is an Optimal Algorithm:  * Stay Ahead Method: Argue that at each step, our algorithm is making more progress than any other algorithm. * Claim: Let be requests returned by EAF, sorted by finish time. Let be any other set of compatible requests, sorted by finish time. For all , . | |

|  |  |
| --- | --- |
| Lecture 14 | 2024-10-07  Week 6, Mon |
| EAF Analysis (Stays Ahead Method):  * Let be the set of requests returned by EAF. * Let be any other compatible set. * Claim: For all , . * Proof: We will prove this by induction on . * *Base Case*: . Then . By construction, we construct the earliest finish time. * *Induction Hypothesis*: Assume for all that . * *Induction Step*: We want to show that as well. * We know that since is a compatible set. Then, by the induction hypothesis. * so is compatible with , but is a compatible request with the EAF, so . * Theorem: EAF returns an optimal set. * Proof: We will prove this by contradiction. * Suppose that , i.e., . * Look at . We know that . * is compatible with , but then EAF wouldn’t have terminated.  The Communication Network Problem:  * Build a communication network at the minimum cost. * **Def:** A spanning tree of a graph is a subset of edges such that is connected and has no cycles. * Cost of spanning time: . * Problem: MinimumSpanningTree (MST). * Input: Graph with edge costs . (Assume edge costs are distinct.) * Output: Minimum-cost spanning tree (MST).      Kruskal’s Algorithm (Kruskal ’56): List KruskalsAlgorithm() {  T = [];  repeat until |T| = n - 1 {  add cheapest edge to T that does not create a cycle;  }  return T;  } Prim’s Algorithm (Prim ’57, Jarník ’30, Dijkkstra ’56): List PrimsAlgorithm() {  start at s;  S = [s];  T = [];  repeat until |T| = n - 1 {  pick cheapest edge (u, v) with u in S, v not in S;  add (u, v) to T;  add v to S;  }  return T;  } | |

|  |  |
| --- | --- |
| Lecture 15 | 2024-10-09  Week 6, Wed |
| Cut Property:  * Let be any nonempty proper subset of the vertices. * Let be the cheapest edge with one endpoint in , one endpoint not in . Then must be in the MST.  Proving Prim’s Algorithm:  * Claim: Prim’s algorithm returns an MST. * Proof: * At each iteration, we have . * Cut property says that the cheapest edge out of is in MST. Note that this is the exactly the edge we add. * This is true for every edge we add in Prim’s algorithm. * Therefore, Prim’s algorithm returns an MST.  Proving Kruskal’s Algorithm:  * Claim: Kruskal’s algorithm returns an MST. * Proof: * At each iteration, look at edge that we add to . * . * By the cut property, the cheapest edge between must be in MST. * We can say this about any edge we add. * Therefore, is an MST.  Proving the Cut Property:    * Proof by Contradiction: * Fix . Let be the cheapest edge while , . * Let be a spanning tree that doesn’t contain the edge . * Since is a spanning tree, so all vertices are connected. Let be a path in . * Notice that is a cycle. There must be some edge that crosses cut (crosses and ). * Swap for , and we get a new, cheaper spanning tree. | |

|  |  |
| --- | --- |
| Lecture 16 | 2024-10-11  Week 6, Fri |
| **Q:** Among all closed shapes with perimeter, what is the maximum area ?  **A:** . The largest shape possible is a circle.   * We can prove this answer using exchange arguments.  Exchange Argument:  * Motivation: Solutions can be hard to compare, but comparing closer solutions might be manageable.  1. Say that our solution is . 2. Start with any solution . 3. Make small changes to get without making things worse. 4. Repeat this process. 5. Arrive at our solution . 6. Conclude that this solution is optimal.  The Interval Scheduling with Deadlines Problem:  * Input: List of jobs where each job has a time required to complete and a deadline . * Output: A schedule of when to complete all jobs that *minimizes maximum lateness*. * Note: All jobs must be completed. * Notation: For any schedule , let , be the start and finish times of job in schedule . * Def: The lateness of job , , is defined as if . is 0 otherwise. * Goal: Come up with schedule that minimizes .  Possible Greedy Algorithms:  1. Earliest Deadline: . Works 2. Earliest Completion Time: . Does Not Work   Counterexample: , , , . We will pick the first job first which actually results in a larger maximum lateness.   1. Minimize Slack Time: . Does Not Work   Counterexample: , , , . We will pick the second job first which actually results in a larger maximum lateness.   * Theorem: Earliest Deadline returns the optimal schedule. * Def: Idle time exists when there are jobs left to complete but nothing is being done at the moment. * Fact: There is an optimal schedule with no idle time. * Def: An inversion in a schedule is a pair of jobs , such that but is scheduled after . * Fact: Earliest Deadline returns schedule with no inversions. * Def: , is an adjacent inversion if but is scheduled immediately after . * Claim: If schedule has an inversion, then it has an adjacent inversion. * Proof (Via Exchange Argument): * Let be any schedule that has an inversion. Then has an adjacent inversion . * Let be , but with jobs , swapped. * Note: now has one less inversion compared to . * Claim: . | |

|  |  |
| --- | --- |
| Lecture 17 | 2024-10-21  Week 7, Mon |
| **Q:** What is a greedy sorting algorithm?  **A:** Selection sort. In each step, we take the next smallest element, and we put it into place. Divide and Conquer:Merge Sort: Array MergeSort(Array A) {  Divide A into 2 halves L, R;  MergeSort(L);  MergeSort(R);  Merge(L, R);  }   * Divide and Conquer Algorithms:  1. Divide input into two (or more) pieces of equal size. 2. Solve each subproblem. 3. Combine results into overall solution.  Runtime of Merge Sort:  * Let denote the time needed to merge sort a list of elements. Then, * for and some constant . . * Techniques for solving recurrence relations: * Substitution method. * Recursion trees.  Substitution Method:  1. Guess solution: . 2. Substitute into equation: . 3. Try to prove by induction. 4. Repeat steps 1-3 until we get the correct solution.  * Claim: . * Proof: We will attempt to prove this by induction on . * *Base Case*: . . Notice . The base case is correct. * *Induction Hypothesis*: Assume for all . * *Induction Step*: We want to show for as well. * From the recurrence relation, we know that . We can substitute in using our induction hypothesis where . We get . We now have . Note that .  Recursion Trees (Unrolling the Recursion):  1. Analyze the first few levels. 2. Try to identify the general pattern. 3. Look at the amount of work at the last level. 4. Sum up all the work over all levels.  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | Level | Problem Size | # of Nodes | Work Per Node | Total Work | | 0 |  | 1 |  |  | | 1 |  | 2 |  |  | | 2 |  | 4 |  |  | |  | | | | | |  |  |  |  |  | | (Leaf) |  |  |  |  |  * Notice that from the final problem size, we have and . * The total amount of work is . * Therefore, .  Recurrence Relations:Intuition:  * The runtime for any recurrence relation depends on 3 factors:  1. The branching factor (# of recursive calls). 2. The change in problem size. 3. The work per node. | |

|  |  |
| --- | --- |
| Lecture 18 | 2024-10-23  Week 7, Wed |
| Similarity Ranking:  * Ranking: A sequence of distinct numbers . For example, movie 1 is my th favorite, movie 2 is my th favorite. * Inversion: Given two rankings and . An inversion is a pair such that but .  Inversion Counting Problem:  * Counting the number inversions. * Simple version: Assume sorted. For example, . * Input: Ranking . * Output: The number of inversions that belong to and . I.e., the number of pairs such that and .  Exercise:  * Design algorithm for ranking similarity. Produce pseudocode.  Brute-Force Algorithm: int BruteForceCount(Array a) {  int count = 0;  for(i in range(1, n)) {  for(j in range(i + 1, n + 1)) {  if(a\_i > a\_j) {  count += 1;  }  }  }  return count;  } Divide and Conquer Algorithm:  * Divide into 2 halves. * Count the number of inversions in the left half. * Count the number of inversions in the right half. * Count the number of inversions between halves.   Tuple MergeCount(Array L, Array R) {  count = 0;  i, l, r = 0;  s = L.size();  while(L, R not exhausted) {  if(L[l] < R[r]) {  A[i] = L[l];  l += 1;  }  else {  A[i] = R[r];  r += 1;  count += (S - l);  }  i += 1;  }  // Add in remaining items from subarray.  return (A, count);  }  Tuple SortCount(Array A) {  // Return sorted list.  // Also return the number of inversions in A.  if(A.size() <= 1) {  return (A, 0);  }  else {  Divide A into 2 halves L, R;  (L, C\_L) = SortCount(L);  (R, C\_R) = SortCount(R);  (A, C) = MergeCount(L, R);  return (A, C\_L + C\_R + C);  }  }  **Q:** What is the runtime?  **A:** , . . Example:  * . * Number of inversions involving the right-hand side elements: * 5:2, 6:2, 1:4, 2:4. | |

|  |  |
| --- | --- |
| Lecture 19 | 2024-10-25  Week 7, Fri |
| The Integer Multiplication Problem:  * Inputs: 2 -digit, base- numbers and . * Output: .  Elementary School Multiplication: int ESMultiplication(int a, int b) {  // Precompute N \* N timestable.  int sum = 0;  for(i in range(0, n)) {  for(j in range(0, n)) {  sum += b\_i \* a\_j \* N^(i + j);  }  }  return sum;  }   * Operations involved: * -digit addition: . * Single-digit multiplication (e.g., ): . * -shifting: . * Traditionally, multiplication is since we need to multiply every single digit of with every single digit of (and moving and adding zeros appropriately).  Divide and Conquer for Multiplication:  * . * . * . * For this algorithm, we split into 4 different multiplications each step. Therefore, , . * Solving this recurrence relation, we get . * Why didn’t we get an improvement? We have too many (4) recursive calls in each step. * Observe . This contains the exact 4 terms we need in one multiplication, so we can subtract and from . This results in getting and all terms needed in 3 instead of 4 multiplications.  Karatsuba Multiplication:  * . * . * . * (Note that this is ).   int Karatsuba(int a, int b) {  int A = Karatsuba(a\_L + a\_R, b\_L + b\_R);  int B = Karatsuba(a\_L, b\_L);  int C = Karatsuba(a\_R, b\_R);  return (B \* N^n + (A - B \* C) \* N^(n / 2) + C);  }   * Overall runtime; , .  Partial Substitution for Karatsuba Multiplication:  * Guess: for constants , , and to be determined. * Proof: We will attempt to prove this by induction on . * *Base Case*: For , and . The base case holds as long as . * *Induction Hypothesis*: Assume for all that . * *Induction Step*: . * Notice that this last inequality holds if and . * For , we have . So we set . * For , we have . | |

|  |  |
| --- | --- |
| Lecture 20 | 2024-10-28  Week 8, Mon |
| The Steel Rod Problem:  * Inputs: representing the length of rod and a list of prices where represents the revenue from selling an feet rod. * Output: Maximum possible revenue from chopping feet rod into pieces, selling the pieces.  Greedy Algorithms for Steel Rod:  1. Cut rod to get price that maximize revenue, then repeat with the remaining rod. Does Not Work   Counterexample: , , , , for all other . The greedy algorithm outputs 2 of , equaling 10, while the optimal is 5 of , equaling 20.   1. Cut rod to get price with maximum revenue per foot, then repeat with the remaining rod. Does Not Work   Counterexample: , , , , for all other . The greedy algorithm outputs 1 of and 4 of equaling 16, while the optimal is 2 of equaling 20. Divide and Conquer Algorithm for Steel Rod:  1. Cut the rod in half, recurse so that . Does Not Work   Counterexample: , , , for all other . The greedy algorithm outputs , while the optimal is . This algorithm just doesn’t make sense. Brute Force Algorithm for Steel Rod:  1. Try all the possible ways to cut a rod. Then calculate the revenue for each method. Works But Inefficient   Problem: There are an exponential number of possible solutions. For each rod (assuming integer cuts), we have places to cut, which leads to possible solutions to go over. Dynamic Programming:  * Solve problem by identifying a collection of subproblems, solving subproblems one-by-one, using solutions to smaller subproblems to understand the larger ones.  Dynamic Programming Design Process:  1. Characterize the structure of the optimal solution. 2. Recursively define the value of the optimal solution. 3. Compute the value of the optimal solution. 4. (If needed,) construct the optimal solution from the computed information.  Dynamic Programming for the Steel Rod Problem:  1. Let be the maximum revenue we can get from an -feet rod. If it is optimal to cut the rod at feet, then .   (*Alternative Characterization*) If the leftmost cut in the optimal solution is at feet, then .   1. . Define . 2. See the pseudocode below.   int SteelRod(int n, list P) {  if(n == 0) {  return 0;  }  int q = -1;  for(int i = 1; i <= n; i++) {  q = max(q, P[i] + SteelRod(n - i, P));  }  return q;  }  **Q:** What is the runtime?  **A:** , . In the end, . | |

|  |  |
| --- | --- |
| Lecture 21 | 2024-10-30  Week 8, Wed |
| The RNA Substructure Problem:RNA Molecule:  * A string of bases , where each .  Secondary Structure on :  * A set of pairs such that:  1. (No sharp turns.) Each pair has . 2. Elements of any pair must be , . 3. must be a matching. 4. (No crossing.) If and , we can’t have .  * Big Question: What secondary structure is most likely to form. * Big Answer: The one that minimizes total free energy. (CS Translation: Maximize the size of matching .)  The Problem:  * Inputs: The RNA molecule . * Output: ① The size of the largest possible secondary structure and ② itself.  Dynamic Programming Step 1:  * Characterize the structure of the optimal solution. * the maximum number of pairs in string . * Choice: Does get paired? If so, with what? * not paired: the score is . * paired: say is paired with , the score is 1 for + for + ??? for . * Notice that we don’t have a good way to characterize the score for . Therefore, this is not the right structure. | |

|  |  |
| --- | --- |
| Lecture 22 | 2024-11-01  Week 8, Fri |
| The RNA Substructure Problem (Continued):Dynamic Programming Step 1, Attempt 2:  * Let be the size of the best matching for substring . * In the optimal matching, is either going to get matched or not. * If is not matched, then . * If is matched to , then .  Dynamic Programming Step 2:  * Define a 2D table where stores . * Note: We want to output . * General Case: . * The term means that to take over all the ’s that are valid matches for (valid base-pairing and ).  Memorization:  * Store results of previous function calls. * Return cached results if some function call is called again.  Dynamic Programming Step 3: List RNA(List B) {  Initialize T[i, j] = -1 for all i, j;  return RNAH(B, T, 1, n);  }  List RNAH(B, T, i, j) {  if(|i - j| <= 4) {  return 0;  }  if(T[i, j] >= 0) {  return T[i, j];  }  // Computing T[i, j].  q = RNAH(B, T, i, j - 1);  for(t = i, ..., j - 5) {  if(B[t], B[j] are valid matches) {  temp = 1 + RNAH(B, T, i, t - 1) + RNAH(B, T, t + 1, j - 1);  if(temp > q) {  q = temp;  }  }  }  T[i, j] = q;  return T[i, j];  } Example:  * In the case of : * . * Notice that in the equation above, we are assuming that can be matched with . | |

|  |  |
| --- | --- |
| Lecture 23 | 2024-11-04  Week 9, Mon |
| Memorization:  * “Top-down” dynamic programming implementation. * Recursively compute problems when needed. * Cache stored solutions to subproblems so they don’t need to be recomputed.  Tabulation:  * “Bottom-up” dynamic programming implementation. * Compute the smallest subproblems first, then using them to compute larger subproblems.  The RNA Substructure Problem (Continued):Tabulation for the RNA Substructure Problem:  * Let be a 2D table with all entries initialized to 0.   List RNA(B) {  Initialize T[i, j] = 0 for all i, j;  for(k = 5, ..., n) {  for(i = 1, ..., n - k) {  j = i + k;    // Computing T[i, j].  q = T[i, j - 1]  for(t = 1, ..., j - 4) {  if MATCH(B[t], B[j]) {  score = 1 + T[i, t - 1] + T[t + 1, j - 1];  if(score > q) {  q = score;  }  }  }  T[i, j] = q;  }  }  return T[1, n];  }   * Notice that we should check for whether or not the indices are valid when we index previous table entries.  Memorization vs. Tabulation:  * Memorization: For problems where the numbers needed in the table entries is less than the total number. If so, memorization is much faster as we don’t need to compute every possible entry through recursion. * Tabulation: When most or all of the subproblems are needed, tabulation is usually constant-factor faster. Why? Recursion calls need to set up and manage a new stack frame which is not needed for iteration.  Generalization Dynamic Programming Implementation Details:  * We can use an array or dictionary for tables. * We have a surprisingly large -factor overhead for dictionary. * Watch out for edge cases (e.g., negative table indices). * Design: Often the problem constraints can be handled at the expense of having extra dimensions in the table. The table can store int/float, but also bool. We can also perform postprocessing when needed.  Dynamic Programming Step 4:  * For the RNA substructure problem, our pseudocode returns the size of the optimal matching, not the matching itself. We have two ways of constructing the solution. * Create a second table . We now have two options:  1. Each entry in stores a partial solution to that subproblem (e.g., stores the optimal matching for for the RNA substructure problem). 2. Store in the choice which gives us the optimal solution. Perform postprocessing to recover the optimal solution.  Example for Option 2: List RNA(B) {  Initialize T[i, j] = 0 for all i, j;  Let S[0 ... n][0 ... n] be a 2D table storing what j should be matched with in the optimal matching.  for(k = 5, ..., n) {  for(i = 1, ..., n - k) {  j = i + k;    // Computing T[i, j].  q = T[i, j - 1]  for(t = 1, ..., j - 4) {  if MATCH(B[t], B[j]) {  score = 1 + T[i, t - 1] + T[t + 1, j - 1];  if(score > q) {  q = score;  S[i, j] = t; // The best matching for B[i] ... B[j] that matches (t, j).  }  }  }  T[i, j] = q;  }  }  return T[1, n];  }   * Then perform postprocessing where we make recursive calls to rebuild the matching. | |

|  |  |
| --- | --- |
| Lecture 24 | 2024-11-06  Week 9, Wed |
| Network Flow:  * Say is a directed graph where each edge has an integer capacity . * There are two vertices . * Think of the integer capacity as the size of the water pipes (edges). * A flow is a function , describing how much flow is on each edge such that:  1. Flow is conserved. For all ( not equal to or ), must equal . 2. Capacity is respected. For all , must have .  * Def: The value of a flow is .    The Network Flow Problem:  * Inputs: * A directed graph . * Edge capacities . * . * Outputs: The maximum flow (or the value of ). * **Q:** Does ? * **A:** Yes, because flow is conserved.  Residual Graphs:  * . * For each edge , add an edge to with capacity (“forward edge”). * For each edge with , add a reverse edge to with the capacity (“backwards edge”). * Def: Given and a flow , the residual graph is agraph where and with 2 kinds of edges: * Forward edges: For any , add with . * Backward edges: For any with , add with . | |

|  |  |
| --- | --- |
| Lecture 25 | 2024-11-08  Week 9, Fri |
| Residual Graph Examples:  |  |  | | --- | --- | | **Flow Graph** | **Residual Graph** | |  |  |  The Ford-Fulkerson Algorithm: function FordFulkerson(Graph G) {  Initialize f(e) = 0 for all edges;  Initialize residual graph G\_f;  while(there is augmenting s -> t path P in the residual graph) {  f' = augment(f, P);  f = f';  Recompute the residual graph G\_f with new flow;  }  return f;  }  int bottleNeck(function f, Path P) {  return minimum edge capacity of edge in P in G\_f;  }  function augment(function f, Path P) {  b = bottleNeck(f, P);  for(edge e = (u, v) in P) {  if((u, v) is a forward edge):  f((u, v)) += b;  if((u, v) is a backward edge):  f((v, u)) -= b;  }  return f;  } Questions:  * Does FF halt? * Does FF run in polynomial time? * Does FF return a valid flow? * Is the returned flow maximal?  Halting Proof:  * Claim: FF algorithm halts. * Initial flow has value . * Fact: For any flow , . * Every iteration of FF, we find augmenting path, i.e., in residual graph where every edge has nonzero capacity. * Value of flow increases in bottleneck > 0.  Runtime Analysis:  * For any flow , creating residual graph can be done in time. * Use BFS to find augmenting paths can be done in time. * Other work inside the while loop is for iterations. * Therefore, the overall runtime is . | |

|  |  |
| --- | --- |
| Lecture 26 | 2024-11-11  Week 10, Mon |
| The Ford-Fulkerson Algorithm (Continued):  * Claim: The Ford-Fulkerson algorithm returns a valid flow. * Proof Sketch: * Claim: For all , after iterations of while loop, we have a valid flow. * Proof: By induction on . * *Base Case*: Our flow is zero everywhere, so this is definitely a valid flow. * *Induction Hypothesis*: Assume claim holds for all . * *Induction Step*: We want to show that this holds for as well. By the induction hypothesis, we start with a valid flow . Say an augmenting path. Look at any vertex on the augmenting path: * *Case 1*: We have 2 forward edges. In this case, we can increase flow into and out of . * *Case 2*: We have a forward edge and a backward edge. In this case, increasing flow on the forward edge causes the exact decrease on the backward edge. * *Case 3 and Case 4* are similar.  Cut:  * Def: An cut is a partition of into two sets such that . * The capacity of cut() is . * Lemma 1: For any cut and any flow , . * Lemma 2: For any flow and any cut , . * Theorem: The Ford-Fulkerson algorithm returns a maximum flow. * Proof Sketch: Cleverly choose cut so that for flow returned by the Ford-Fulkerson algorithm, .  Proof of Lemma 1:  * Suppose . * . * Notice that the last equality is true since since flow is conserved. * .  Proof of Lemma 2:  * By Lemma 1, we have  Proof of Cut:  * Let be the flow returned by the Ford-Fulkerson algorithm. * Let {vertices such that there is an augmenting path in }. * Let {vertices such that there is no augmenting paths}. * Claim: . * Notice that any forward edge from to in has capacity 0 in . Therefore, in the flow graph, . * In other words, any backwards edge in with , has capacity 0 in . Thus, there is no flow in the flow graph and . * . * Notice that . * . | |

|  |  |
| --- | --- |
| Lecture 27 | 2024-11-13  Week 10, Wed |
| Computational Complexity Theory:  * The classification of problems occurring to their inherent difficulty. * : The set of all decision problems solvable in polynomial time. * Def: is polynomial-time reducible to (write ) if arbitrary instances of problem can be solved using a polynomial amount of time, plus a polynomial number of solutions to . * Note: “” means that “ is polynomial-time reducible to ” or “reduce from problem to problem ”. * Consequences of :  1. If can be solved in polynomial time, then so can . 2. If cannot be solved in polynomial time, then neither can .  Some “Hard” Problems:  * Hard in quotes as we don’t have rigorous proofs for an inherent lack of polynomial-time solutions for these problems.  1. Satisfiability (SAT):  * Input: boolean variables ; clauses where each clause is OR of some variables or their negation. * Output: YES if there is a satisfying assignment to . (I.e., if we can assign values so that every clause evaluates to TRUE.)  1. 3-Color (Tripartiteness):  * Input: Graph . * Output: YES if we can color vertices using one of 3 colors so that for each edge, endpoints are different colors. * Def: An independent set is a set of vertices such that for all , . * Def: A vertex cover of a graph is a subset of vertices such that for all , either or or both (every single edge has an endpoint in ).  1. Independent Set (IS):  * Input: Graph , integer . * Output: YES if and only if has an independent set of at least vertices.  1. Vertex Cover (VC):  * Input: Graph , integer . * Output: YES if and only if has an vertex cover of size .  Theorem:  1. . 2. . | |

|  |  |
| --- | --- |
| Lecture 28 | 2024-11-15  Week 10, Fri |
| * Lemma: Let be a graph. Then is an independent set if and only if is a vertex cover. * Proof: Let . If is a vertex cover, then every edge has some endpoint in . * Notice that it can’t be the case where there is an edge with both endpoints in . * Therefore, is an independent set. * If is an independent set, then any edge has at least one endpoint not in . I.e., or . * Therefore, is a vertex cover. * Corollary: is an independent set of size . is a vertex cover of size .  Polynomial Reduction Examples:  * Theorem 1: .   set IS(Graph G, int k) {  return VC(G, n - k);  }   * Theorem 2: .   set VC(Graph G, int k) {  return IS(G, n - k);  } The Set Cover Problem:  * Input: A set of elements. is “the universe”; subsets of : ; an integer . * Output: YES if and only if there is a set such that and . * Theorem 3: . * Proof: Let and be the inputs for the VC problem. For each vertex , define . .   set VC(Graph G, int k) {  for(int i = 1; i <= n; i++) {  S\_i = {edges e with i as endpoint};  }  return SC(E, S\_1 ... S\_n, k);  } | |

|  |  |
| --- | --- |
| Lecture 29 | 2024-11-18  Week 11, Mon |
| * Claim 1: If has a set cover with , then has a vertex cover of size . * Claim 2: If there is a vertex cover of size , then there is a set cover of size . * Proof of Claim 1: (Claim 2 is similar by construction.) Let be a set cover with size . Then for all , there is an such that *.* For each edge , there is such that . I.e., has as its endpoint. Let . * Then for all edges, one of its endpoints is in . Thus, is a vertex cover. * Nuance: Pay attention to the problem size.  The Satisfiability Problem (SAT):  * Inputs: Boolean variables ; clauses . * Literal: A variable or its negation  . * Each clause is OR of literals. For example. . * Outputs: YES if and only if we can assign truth values to variables so that each clause is satisfied, i.e., evaluates to TRUE.  The 3-SAT Problem:  * Like SAT, but each clause is OR of three literals. * More generally: -SAT. Each clause is OR of literals. * Fact: For any ,-SAT SAT. * Fact: For any ,SAT -SAT. * Fact: There is a polynomial-time algorithm for 2-SAT.  Polynomial Time Verifiers:  * is a polynomial-time verifier for a decision problem if:  1. takes two inputs , . 2. There is a polynomial function such that for all strings , if and only if there is a string with and YES.  Important High-Level Considerations:  * The length of is polynomial in the length of . * If is YES input for , there is some which causes to output YES. * If is NO input, should output NO no matter what is.  Verifier for the 3-Color Problem:  1. Convert into graph . Let be the number of vertices in the graph. 2. If is not a bit string, return NO. 3. Treat as an array of colors.  * We define the following encoding for with a length bits: 00 for red; 10 for blue; 01 for green; 11 is not used.  1. If for any , return NO. 2. For each edge , return NO if . 3. Return YES.  Verifier for the Not-Factoring Problem (Lab Problem 6):  * . * represents prime factorization of . .  1. Validate format of . 2. Check that for all . 3. . 4. Verify that each is prime. | |

|  |  |
| --- | --- |
| Lecture 30 | 2024-11-20  Week 11, Wed |
| The 3-SAT Problem:  * Inputs: Boolean variables ; clauses , each clause is an OR of 3 literals ( or ), e.g., ). * Output: YES if there is a satisfying assignment.  Reductions with Gadgets:  * When reducing , convert each piece of input in problem A into a piece of input in problem B. * Theorem: 3-SAT IS. * Proof: Let , be a 3-SAT input.  1. For each clause , create a triangle. 2. Add an edge between any pair of conflicting literals (e.g., ).      1. Run IS algorithm on constructed graph and . 2. Return whatever IS algorithm returns.  * Claim: A 3-SAT input is only satisfiable if and only if graph has IS of vertices. * Proof (Forward Direction): Suppose a 3-SAT input is satisfiable, then fix a satisfying truth assignment for clauses. We can pick one literal in each clause that evaluates to TRUE. The vertices corresponding to these literals form an independent set in the graph. * Proof (Reverse Direction): If a constructed graph has an IS of size , this IS has no conflicting literals. We can find a satisfying truth assignment for the 3-SAT input by setting variables so that all literals in the IS evaluate to TRUE and setting arbitrary for the rest of our assignment. * Theorem: Prove if and , then . * Proof Sketch: Take an algorithm for A that uses B as a subroutine. Instead of calling the subroutine for B, emulate the algorithm for B that uses C as a subroutine. * To-do: Argue that we do polynomial amount of work plus a polynomial number of calls to C.  SAT Reducibility Chain:The Cook-Levin Theorem:  * Every decision problem in the complexity class NP can be reduced to the SAT problem. * If we can solve the SAT problem in polynomial time, then P = NP. | |

|  |  |
| --- | --- |
| Lecture 31 | 2024-11-22  Week 11, Fri |
| Complexity Classes:  * P: The set of all decision problems solvable in polynomial time. * NP: The set of all decision problems verifiable in polynomial time. * Fact: . * NP-Hard: The set of all decision problems such that for any , . * For example, the Cook-Levin theorem tells us that SAT is in NP-Hard. * NP-Hard does not intersect P. Also contrary to naming, NP-Hard . * NP-Complete: NP NP-Hard. * There are polynomial-time verifiable problems to which any problem in NP can be reduced. These seem to be the key to finding out whether P = NP, or .   undefined Showing a Problem is NP-Complete:  1. Show . 2. Pick problem known to be NP-Complete. 3. Reduce . Given an instance for problem , construct (in polynomial time) instance for such that: 4. If is YES input, then so is . 5. If is NO input, then so is .  * Theorem: 3-Color is NP-Complete. * Proof: ① 3-Color NP. ② Pick 3-SAT. ③ Reduce 3-SAT 3-Color using *gadgets*.  3-Color Graph Coloring:  * For each problem, create a 3-colorable graph with exactly that property and no other constraints. You can add vertices you want, and can hardwire vertices to be red, blue, or green.  |  |  |  | | --- | --- | --- | | 1. , , all have different colors. | 1. , , do not all have the same color. | 1. None of , , is green and they cannot all be blue. | | 1. , , all have the same color. | 1. None of , , is green. |  3-SAT → 3-Color Reduction:  1. Create a triangle with vertices T, F, and B. 2. For each Boolean variable , create a triangle between , , and B.      1. For each clause (e.g., ), create a gadget as shown in problem 5 in the previous section.      1. Connect everything together. For any nodes with the same label, make sure that they have the same color.     Note that vertices and in the graph above will have the same color. | |

|  |  |
| --- | --- |
| Lecture 32 | 2024-11-25  Week 12, Mon |
| Approximation Algorithm:  * Def: An approximation algorithm is an algorithm that runs in polynomial time and produces solutions that are guaranteed to be to be close to optimal. * Note that approximation algorithms still must produce the correct answer. * Approximation Ratio: An algorithm has an approximation ratio of (“Rho” of ) if for any input of size , the “cost” produced by the algorithm is within a factor of of the “optimal cost” OPT. I.e., . * Note: If a problem is a minimization problem, then , so . * Note: If a problem is a maximization problem, then , so .  Approximation Ratios:  * OK: . * Better: . (Often constant-factor approximations are not hard to get.) * Best: for any .  The VC-OPT Problem:  * Input: Graph . * Output: Smallest possible vertex cover.  Algorithm VC-Approx:  * Provided a graph .   set VCApprox(Graph G) {  set C <- {};  E' <- E;  while(E' != {}) {  pick edge (u, v) in E';  add u, v to C;  remove from E' all edges incident to u or v;  }  return C;  }   * Theorem: VC-Approx is a 2-approximation algorithm. * Proof: Let be the optimal (i.e., smallest) vertex cover for . * Let . Be the set of all edges picked by our algorithm in the first line in the while loop. Notice that the edges in share no endpoints (they are endpoint disjoint). * Therefore, since each edge in must be covered by different vertices. Also notice that the size of our covering is . Therefore, , so . * For the vertex cover problem, the approximation resistance applies for any approximation smaller than 2. Finding such an approximation that is also efficient requires .  The Weighted Vertex Cover (WVC) Problem:  * Input: Graph with vertex weights . * Output: Minimum weight vertex cover. For any set . * Theorem: WVC is NP-Hard. * Proof: VC-OPT WVC. Given , assign weight = 1 for all vertices.  MTSP Approximation Algorithm (Lab Problem 1):  1. Compute the MST (minimum spanning tree). 2. Pick an arbitrary start vertex. 3. Perform DFS on the MST. 4. Return the vertices in order of when we visit in our DFS, and add the start vertex at the end. | |

|  |  |
| --- | --- |
| Lecture 33 | 2024-11-27  Week 12, Wed |
| The Weighted Vertex Cover (WVC) Problem (Continued):The Pricing Method:  * The pricing method, aka, the “primal-dual” approach. * Edge will pay to be covered. Vertex wants to be paid to provide coverage. * Fair Pricing: A price is unfair if for any vertex , . * We call a vertex tight if . * Note: Let be the optimal weighted vertex cover, and suppose prices are fair, then .  Algorithm: set WVCApprox(Graph G, set {w\_i}) {  Initialize P\_e = 0 fo all edges;  while(there is edge e = (i, j) such that neither i nor j are tight) {  Fix such an edge e;  Increase P\_e until either i or j is tight;  }  Return set S of tight vertices;  }   * Claim 1: is a vertex cover. * Proof: If is not a vertex cover, then there is some edge such that neither nor is covered. is a set of tight nodes, so this means neither nor are tight. But then we wouldn’t have stopped the while loop. * Claim 2: . * Proof: . The second equality is true because all are tight. The third equality is true because each edge is incident to only 2 vertices, so each can appear at most 2 times. | |

|  |  |
| --- | --- |
| Lecture 34 | 2024-12-02  Week 13, Mon |
| Linear Programming (LP):Example:  * Goal: Minimize (the objective function). * Subject To: ; ; (constraints). * The space that satisfies all the constraints is called the feasible solution space.  General LP Form:  * Goal: Minimize the objective function . * Subject To:  1. *Constraints*: ; ; …; . 2. *Nonnegative Variables*: .  Matrix Form:  * . * . * . * We want to minimize subject to and .  History of LP:  * Motivation: Optimization and planning. * [Kantorovich ’39, Koopmans ’42]: WWII planning and logistics; 1975 Nobel Economics Prize. * [Dantzig ’47]: A simple method that is efficient in practice, but exponential time in the worst case. * [Khachiyan ’81]: LP P. The ellipsoid method.  Weighted Vertex Cover as a ~~Linear~~ Integer Program:  * For each vertex, create a variable , where if and if . * Goal: Minimize . * Subject To:  1. *Edge Constraints*: For each edge , add constraint . 2. *Boolean Variables*: Each .  * **Note that this is actually an integer program (IP).** * Integer Program: Like LP, but the variables are restricted to be integers. * Problem: IP is known to be NP-Hard. The fact that integers are discrete make integer programming hard, while linear programming is generally solvable in polynomial time.  LP Relaxations:  * Express an optimization problem as an integer program (IP). * Unfortunately, IP is hard unless P = NP. * Good news: we can solve a related LP is polynomial time, and then use the solution to get a reasonable approximation.  LP Relaxation of WVC:  * Run the following LP: Minimize subject to:  1. for all . 2. for all .  * Output . * Next time, we will show: * Claim 1: is a vertex cover. * Claim 2: , where is the optimal weighted vertex cover. | |