Project 1: Knapsack and Graph Coloring

Zach Neveu

May 7, 2019

1 Knapsack

Due: Monday 5/13

Given: Knapsack Instance
8 <- # of objects
1639 <- # Size bound
0 22 27 <- number, value, size

- Within given time, find best soln. on your computer
- Write function exhaustiveKnapsack(Knapsack &K, int t)
- Use given code to read from files
- Write code to exhaustively run knapsack search
- Classes for items - use `select` functions to select items
- Use `printSolution()` to dump results of knapsack once optimal found
- Submit `resultsX.txt` with results
- Ok to compare results with other groups
- 20-30 instances to solve, 10 mins per piece
- Knapsack is the easy part, get this done!
- Just use brute force

2 Graph Coloring

- Idea: Map of USA, adjacent states can't be same color. How many colors needed? If colors are fixed, what is lowest # of conflicts?

Analysis, Big O and Growth of Functions

Zach Neveu

May 27, 2019

1 Book Keeping

- Reading posted
- Lab 1 available

2 Analysis of Algorithms

Problem: a general description of input parameters and the properties that an optimal solution should have

Instance: a specific example of a problem with all parameters specified

- Example: Given a weighted graph, find the cheapest Hamiltonian Cycle (TSP)
- A "problem" can have many instances

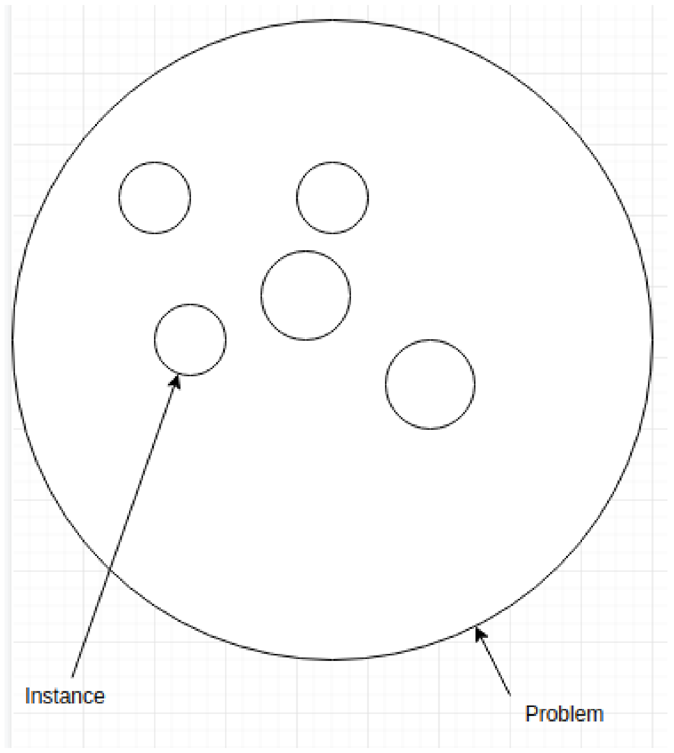
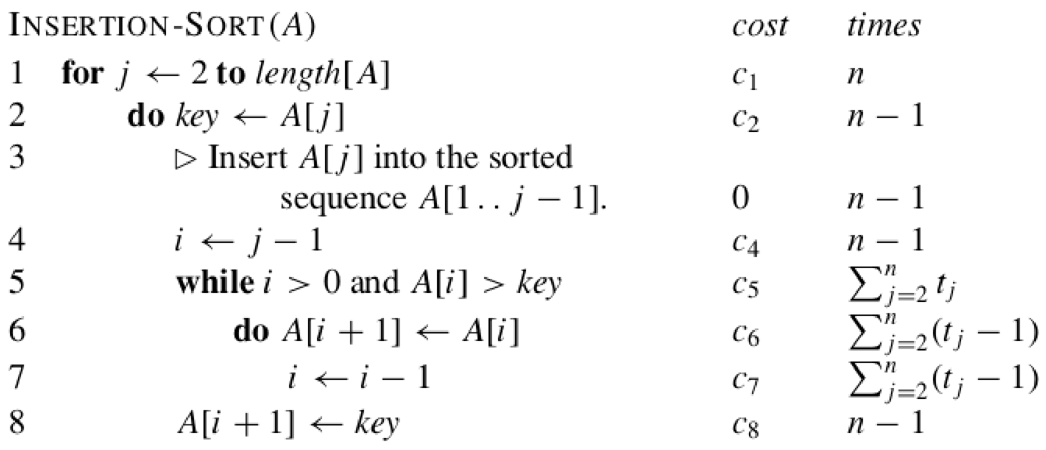


Figure 1: instance_problem

- An algorithm solves all instances of problem

- Many algorithms, what is most efficient?
- What is efficient?
  - Memory
  - Time
  - CPU cycles
  - Disk Space
  - I/O bandwidth
  - Power
- Efficiency usually defined as using smallest time
- Index runtimes by instance size
- "Instance Size" not always well defined - can have multiple params (edges, nodes)

3 Example: Insertion Sort



	INSERTION-SORT(A)	*cost*	*times*
1	**for** $j \leftarrow 2$ **to** $length[A]$	c_1	n
2	**do** $key \leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

Figure 2: Cor p.24

- Best case: already sorted. $T(j) = 1, T(n) = an + b \rightarrow$ linear
- Worst case: reverse sorted: $T(j) = j, T(n) = \frac{n(n+1)}{2} \approx an^2 + bn + c \rightarrow$ quadratic

  Time Complexity Function: The largest amount of time for an algorithm needed to solve the problem for a given instance size.
- Even Time-Complexity function considered too complicated for daily use
- Asymptotic notation used instead

4 Asymptotic Notation

For a given function $g(n)$, $O(g(n)) = f(n)$ there exist positive constants k and n_0 such that $f(n) \leq Kg(n)$ for all $n \geq n_0$
Less formally: $O(g(n))$ is the set of functions that are asymptotically less than $g(n)$ for large n.

Example

I claim that $f(n) = an^2 + bn + c = O(n^2)$. If so, then there should exist positive constants k and n_0 such that

$$an^2 + bn + c \leq kn^2$$
$$a + b/n + \frac{c}{n^2} \leq k$$
$$k = a + 1$$
$$n_0\ is\ intersection$$

Summary

- For insertion sort, worst case runtime (time complexity function) is $an^2 + bn + c$ so the complexity is $O(n^2)$

- Also $O(n^3), O(n^4)$ etc.
- Worst case runtime is $O(n^2)$
- Worst case runtime **itself** is upper bound on run time
- $O(n^2)$ is then an upper bound on the general runtime as well!

  Polynomial-time Algorithm: an algorithm whose time complexity function is $O(p(n))$ for some polynomial $p(n)$

  Exponential-time Algorithm: an algorithm that is not polynomial time

EXPONENTIAL VERY BAD

Day 3: Intro to Comp. Complexity, P & NP

Zach Neveu

May 14, 2019

1 Review

- Huge difference between $O(n^k)$ and $O(k^n)$ (Polynomial vs. Exponential)
- Intractable Problems can only be solved (exactly) with an Exponential time algorithm
- Intractable $\neq$ Unsolvable!!

2 New Stuff: Intractable Problems

How to prove a problem is intractable?

1. If solution has size that is exponential, then any algorithm to find that solution can't be polynomial.
2. If problem can't be solved by any algorithm at all (undecidable) - Halting Problem
3. Certain niche problems with small output that are solvable but intractable

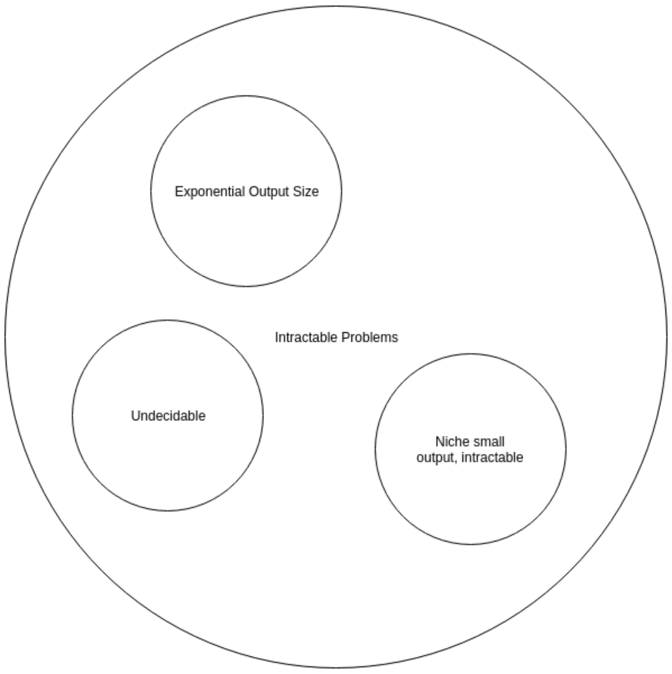


Figure 1: Intractable Problem Categories

3 NP-Completeness

Properties

- No one has ever found a polynomial-time algorithm to solve
- If someone found one algorithm to solve a single NP complete problem in polynomial-time, it would solve all other NP-Complete problems too.
- NP complete problems are either all tractable, or all intractable.
- All in same boat, we just don't know what boat.
- Tons of practical NP complete problems
- Seems unlikely that NP-Complete problems are tractable.
- It is widely believed that the NP-Complete problems are intractable.

What is NP-Complete?

If you can't find an efficient algorithm what do you say?

- "I can't find a solution" - get fired
- "A solution provably doesn't exist" - exceedingly unlikely
- "I can't solve it, but neither can anyone" - show that NP-Complete

Decision vs. Optimization Problems

- Computational Complexity initially developed for decision problems
- Must prove that it can be applied to optimization problems as well
- TSP-opt - Given weighted graph, find shortest Hamiltonian cycle - optimization problem
- TSP-dec - Given weighted graph, is there a Hamiltonian Cycle with weight $\leq k$
- Knapsack-opt - Given objects w/ values and sizes and size bound, find subset to maximize values within size bound.
- Knapsack-dec - Given objects w/ values and sizes and size bound, is there a subset of the objects that are within the size bound and have a value $\geq k$.
- If optimization is tractable, then decision is tractable - just plug in answer from optimization
- If decision is tractable, then optimization is also tractable - just run binary search over k values (adds log of constant)
- Decision and Optimization always have same complexity

4 Complexity Classes

P

- The set of all decision problems that can be solved by a polynomial-time algorithm.
- HC Problem: Given a graph, is it Hamiltonian (does it contain at least one Hamiltonian cycle)?

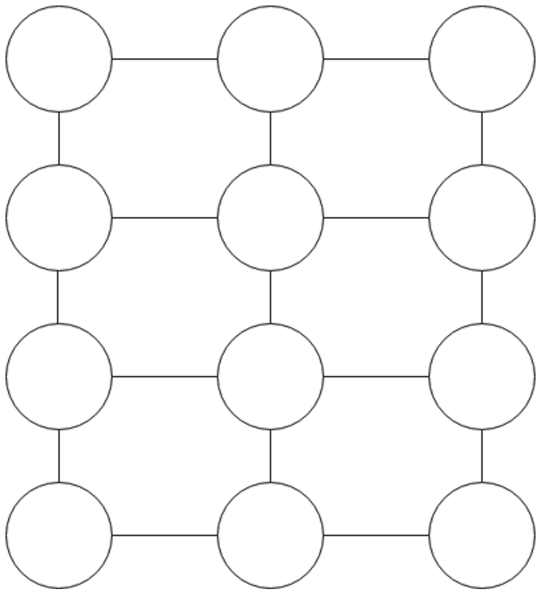


Figure 2: Does this graph have a Hamiltonian Cycle?

- A Verification Algorithm takes some information and checks it to make sure it satisfies the problem requirements.
- Example: given a cycle, verify that it is in fact Hamiltonian
- A **Yes Instance** of a problem will have some certificate that a verification algorithm can verify
- a **No Instance** will not contain a valid certificate to be verified
- Example: TSP-dec
  - certificate: sequence of nodes in the found HC
  - Verification algorithm: Check that each node appears once, edges are valid, and $\sum weights \leq k$
- Example: Matching - given graph and k, is there a matching of size k?
  - Certificate: list of edges in found matching
  - Verification algorithm: make sure no two edges have same end point, edges exist, and number of edges is k
- Shortest Path - given graph, source and dest nodes, and k is there a path from start to end cheaper or equal to k?

  - certificate: ordered nodes in the path found
  - verification: check that edges exist, $\sum weight \leq k$

NP

- Set of all decision problems such that yes instances have a certificate that can be verified in polynomial-time, and no instances do not have a certificate that can be verified in polynomial-time.
- Basically - a verification algorithm exists that works, and it runs in polynomial-time.
- Verification alg. for HC problem runs in polynomial time for yes instances, so HC $\in$ NP
- TSP: also in NP
- Matching: also in NP

Day 4: Reducibility, NP-Completeness, Key Results

Zach Neveu

May 9, 2019

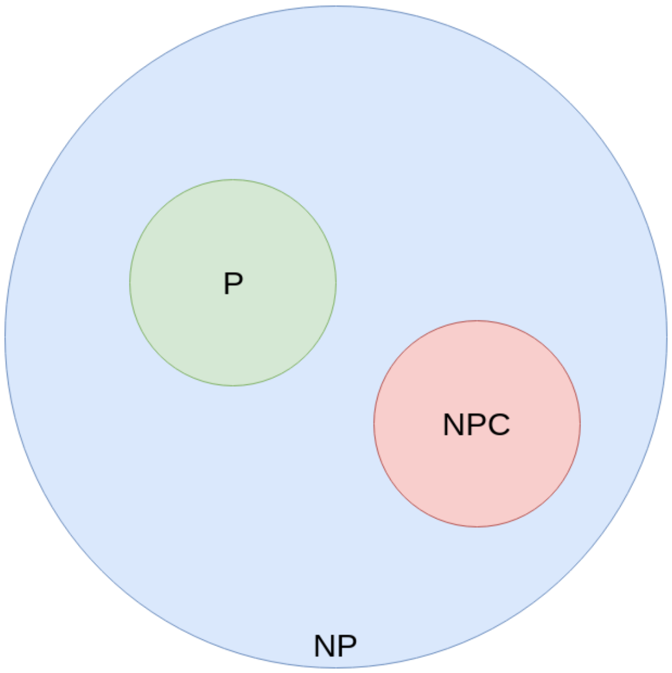


Figure 1: Venn Diagram of P, NP-Complete, and NP

1 Reducibility

Example:
Subset Sum: Given a set of integers and a target, t, is there a subset, S for which $\sum S = t$.
Subset Partition: given a set of integers, can they be partitioned into 2 sets with equal sums?

- If Subset Sum is solved, is it possible to solve subset partition?
- YES! Solve subset sum with $t = \frac{1}{2}\sum S$ where S is all items
- We've just used an SS solver to solve SP! This means that SP reduces to SS.
- If Instance is "no" in SS, it is also "no" in SP

Reducibility: Given problems L_1 and L_2, we say that L_1 is reducible to L_2 in polynomial time if we can rewrite any instance of L_1 as an instance of L_2 such that both instances have the same answer.

Notation: $L_1 \leq L_2$ means that L_1 is reducible to L_2. Starting point, L_1, is on the left. $SP \leq SS$

Example: $HC \leq TSP$
Must be able to rewrite HC as TSP such that they have the same answer.

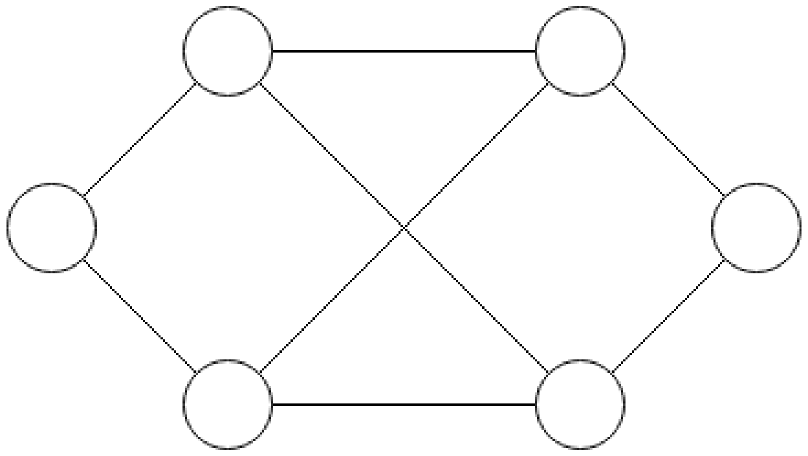


Figure 2: Graph for $HC \leq TSP$ Proof

For proof, must be able to show either:

- A: $yes \rightarrow yes$ and $yes \leftarrow yes$
- B: $yes \rightarrow yes$ and $no \rightarrow no$
- Either A or B requires two steps
- Sometimes one path is much easier
- Option B for $HC \leq TSP$
- If HC is yes instance (HC exists), then the found HC makes TSP a yes instance for weights=1 and bound=num_nodes
- If HC is no instance (no HC exists), then TSP is also no instance because no HCs exist for any cost.

Why is Reduction Useful?

- What if SP is intractable, and SS is in P?
- This is impossible! Reducibility allows you to solve SP in polynomial-time by transforming into SS and solving.

2 NP-completeness

A problem, L, is NP-Complete if:

- $L \in NP$

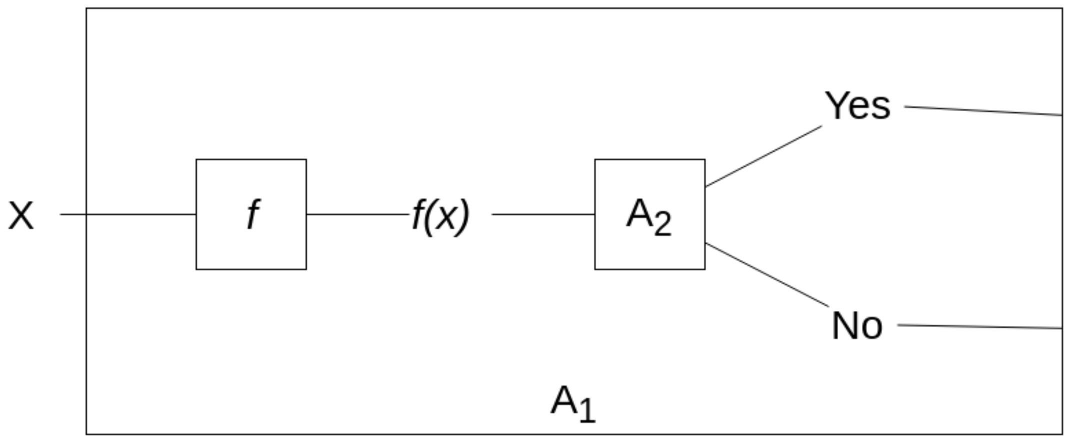


Figure 3: Solving A_1 using A_2 Solver and Reducibility

- For every $L' \in NP$, $L' \leq L$

In words, Every problem in NP should be reducible to L in polynomial time. This essentially means that all NP complete problems are harder than or equal to any other problem in NP. How do we show this?

3 Key Results

1. If $L_1 \leq L_2$, and $L_2 \in P$, then $L_1 \in P$
2. If $L_1 \leq L_2$ and $L_1 \notin P$, then $L_2 \notin P$
3. If L is NPC and $L \in P$, then $NP \in P$
4. If $L' \in NP$ such that $L' \notin P$, then all $NPC \notin P$

4 NPC Examples

Satisfiability (SAT)

- 1971 - Cook found first NPC problem!
- Satisfiability Problem (first one!)
- Consider boolean expression $\overline{x}_3(x_1 + \overline{x}_2 + x_3)$
- Expression is satisfiable if a set of inputs exists which can produce a true output from the expression.
- Given a POS form of an expression, is it satisfiable?
- Ex: $(x_1 + x_2 + x_3)(x_1 + \overline{x}_2)(x_2 + \overline{x}_3)(x_3 + \overline{x}_1)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$

  - Each clause must be satisfiable
  - Going by hand from left to right, we can find that this isn't satisfiable.
- How can every problem be reduced to this?
- All problems in NP have a verification algorithm
- Verification algorithm can be expressed as a satisfiability instance, this is the reduction.
- This shows that $SAT \in NPC$! First problem ever done.
- This result can be leveraged to prove that other problems are NPC
- $NP \leq SAT$

Evolution of Problems

- Year after SAT, first 10 problems shown to be NPC
- After 50 years there are TONS of problems in the list of NPC
- Problems from every field on here.
- When you have a new problem, look for a similar problem that is proved to be NPC and reduce it to your problem.

Arbitrary Problem L_2

- If $L_1 \in NPC$ and $L_1 \leq L_2$ than $L_2 \in NPC$

Day 5 Notes

Zach Neveu

May 21, 2019

1 Agenda

- Review of NPC
- Proving problems NP Complete
- Examples
- Subproblems

2 Announcements

- Quiz on Wednesday - through most recent homework (NOT NPC)
  - Know big ideas
  - Know important terms
  - Know practical applications
  - Re-Solve problems we've seen for practice
  - Make sure we've done the reading
  - No code on quiz
- Finish up reading about NPC stuff

3 NP Completeness Review

To be in NPC, a problem, L, must:

- $L \in NP$
- For every problem $L' \in NP$, $L' \leq L$

How to Prove

- Prove that $SAT \leq L$
- Given a problem $\pi \in NP$ whos complexity is unknown, how to find?

- Define special case π' containing a subset of the instances of π
- Prove that π' is NPC
- $\pi' \leq \pi$ because every special case is already a regular case as well
- π is NPC, since its simpler subset, π' is NPC
- QUIZ: Explain why the last bullet is true

4 Examples

Partition: given a set **A** and size $s(\boldsymbol{A})$ for all $a \in \boldsymbol{A}$ is there a subset $\boldsymbol{A'} \in \boldsymbol{A}$ such that $\sum s(a) = \sum s(!a)$ where $!a$ is the set of elements not in s. Basically, divide A into two sets with equal size.

Knapsack: given a set, U, a size $s(u)$ and a value $v(u)$ for all $u \in U$, and size constraint B, and a value goal K, is there a subset $u' \in U$ such that $\sum s(u') \leq B$ and $\sum v(u') \geq K$?

- Claim: Partition ≤ Knapsack
- Prove: Given an instance of Partition, show that we can produce an instance of knapsack with the same answer.
- Answer: Set $K = B = \frac{1}{2} \sum s(u)$.
- Idea: sandwich K & B such that knapsack will find same answer as partition
- If Knapsack is yes instance, Partition will be yes instance
- If Partition is yes instance, Knapsack is yes instance
- If partition is NPC, Knapsack is NPC

5 Problem as Tuple

- Consider a problem $\pi = (D, Y)$ where D is all instances, Y is all yes instances
- Sub-problem $\pi' = (D', Y')$ reduces to π

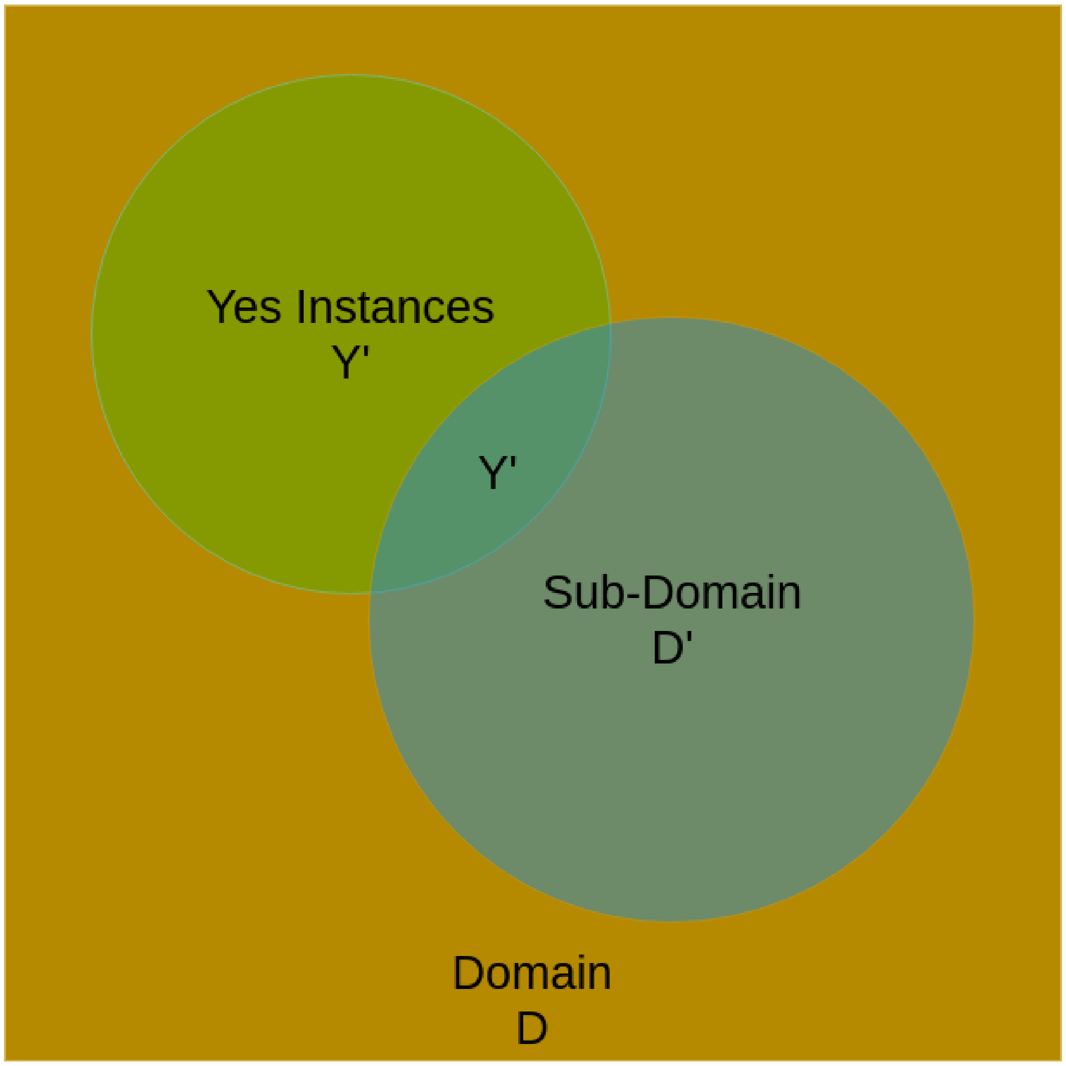


Figure 1: Relation of D, D’, Y, Y’

Day 6 Notes

Zach Neveu

May 14, 2019

1 Review

- NPC proof via sub-problem (see Venn diagram from day 5)
- If sub-problem is NPC, problem is NPC
- If problem is P, sub-problem also P
- Sub-problems can be organized recursively (Complexity Landscape$^{\text{TM}}$)

Example: Procedure Constrained Scheduling (PCS)

- Given a set of tasks that each take one unit of time to complete, a partial order on the tasks, a number, m, of processors, and an integer deadline
- Question: Does a legal schedule allow the processes to be completed before the deadline?
- General PCS ∈ NPC
- If constraints graph is tree, $PCS \in P$
- If constraints graph empty, $PCS \in P$
- Aside: Why m=2 solvable, but m=3 so hard?
- Consider: 3 is important.
- $2SAT \in P$, $3SAT \in NPC$, same for HC problem

Special Nodes

A node on the Complexity Landscape that has no known NPC children is called **Minimally NPC**
A node on the Complexity Landscape that is in P and has no parents in P is called **Maximally polynomially solvable**

2 Greedy Algorithms

- A **Greedy Algorithm** always makes what appears to be the best decision in the current moment
- A **Greedy Algorithm** does not utilize backtracking

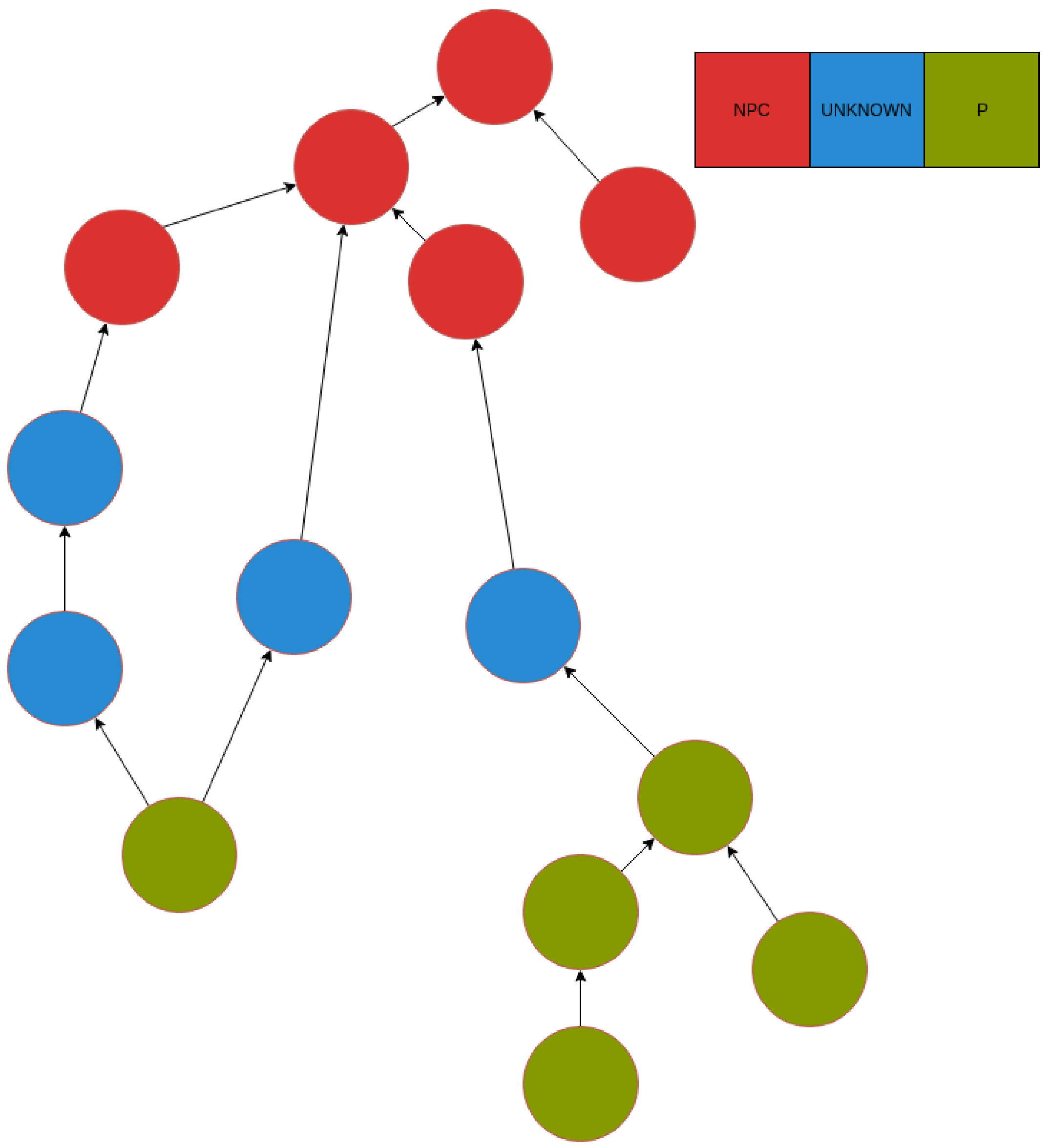


Figure 1: Example Complexity Landscape™

MST

MST: Given an undirected graph and a weight for each edge, find an acyclic subset of the edges that connects all nodes with minimum weight.

```
def MST():
  A=0
  while A is not spanning tree:
    find next edge (u,v) in increasing order by weight such that (u,v) is safe for A
    A += {(u,v)}
  return A
```

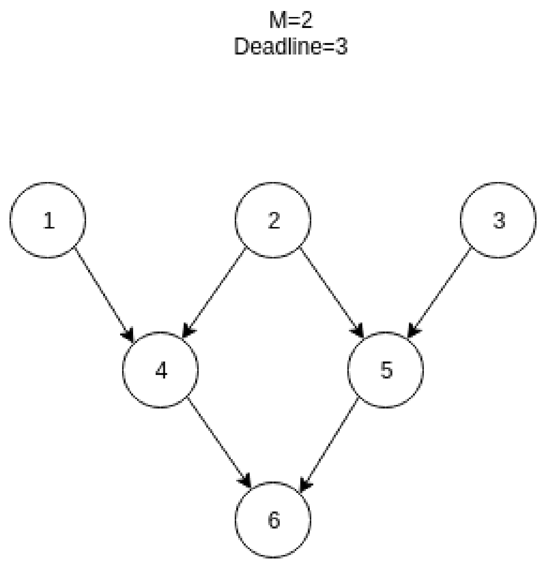


Figure 2: Example PCS Constraint

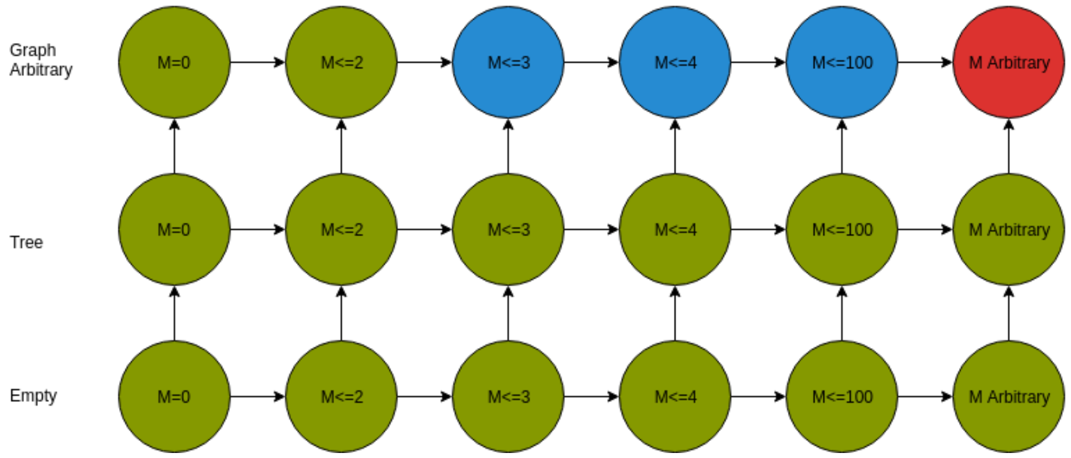


Figure 3: PCS Complexity Landscape^{TM}

Activity Selection

- Given: a set $S = \{a_1, a_2, \ldots, a_n\}$ of n activities, only one of which can take place at a time. Activity a_i starts at time s_i and finishes at time f_i. Two activities are **Compatible** if they do not conflict.
- Find: How many activities can we fit?
- Find a largest set of mutually compatible activities

```
# Solution
# a[i] is most recently selected activity
# a[m] activity we are considering adding
def activitySelection(a):
  sortByIncreaseFinish(a)
  n = number of Activities
  A = a[0] # first activity
```

	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	6	8	8	2	12	3	5
f_i	4	5	6	7	10	11	12	13	14	8	9

Table 1: Example Activity Problem

```
i = 1
for m in range(1,n):
  if not conflict(a[m], a[i]):
    A += a[m]
    i = m
return A
```

- Proof: Show that each step is in the right direction
- Prove by contradiction: assume that there is no maximal subset including a[0]. We can always swap the first event in any set with a[0] because of the sorting, so any maximal subset can include a[0].
- Repeat this problem recursively on all problems with $s_i > f_0$ - this step valid for all sub-problems

Head Partition Problem

- Given directed graph, find largest subset of edges which point to separate nodes

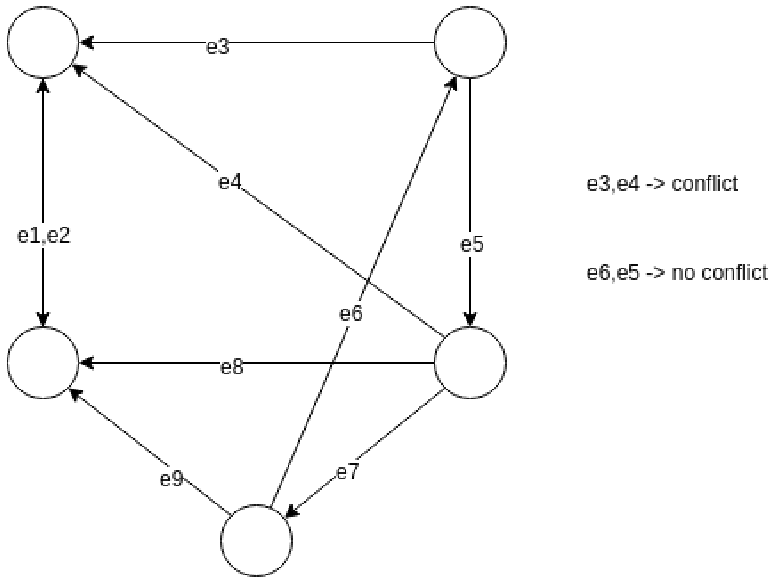


Figure 4: Head Partition Example

```
# Algorithm 1
def headPartition(g):
  for node in g:
    select any incoming arc
```

```
# Algorithm 2
# Version of "Generic Greedy Algorithm"
def headPartition2(g):
  edge_group = {}
  for arc in g.edges:
    if not conflict(arc, edge_group):
      edge_group += arc
```

Day 7 Notes

Zach Neveu

May 15, 2019

1 Agenda

- Quiz
- Greedy Algorithms
- Intro to matching
- Announce next homework
- Announce next project
- Reading

2 Greedy Algorithms

- Classic Case: Minimum Spanning Tree
- Also useful for many other problems
- Single "greedy algorithm" really at the root of all of them

Head Partition (review from Day 6)

- Node-based solution - finds optimal solution
- Edge-based solultion - finds optimal solution, also very similar to MST
  - Go edge-by-edge, add edge if it does not conflict

Generic Greedy Algorithm

```
def generalGreedy(g):
  sort(g.edges)
  soln = {}
  for edge in g.edges:
    if not conflicts(edge, soln)
    soln += edge
```

Weighted Head Partition

Given a weighted, directed graph, find an independent subset with maximum total weight.

- Edge Strategy: Sort edges by decreasing weight. Starting with highest weight edge, add each edge if it does not conflict, else skip.
- Node Strategy: Go through nodes in any order and select the heaviest edge pointing to the given node

Partition

- Let E be a finite set. π is a partition of E if it is a collection of disjoint subsets of E such that the subsets collectively cover E.
- $E = \{e_1, \dots, e_8\}$, $\pi = \big\{\{e_1\}, \{e_2, e_3\}, \{e_4, e_5\}, \{e_6, e_7, e_8\}\big\}$
- $\pi.weights = \big\{\{5\}, \{3,4\}, \{7,8\}, \{2,6,1\}\big\}$
- A subset of E is **Independent** if no two elements come from the same component of π
- $\{e_1, e_4\}$ and $\{e_1, e_4, e_3\}$ are independent
- $\{e_1, e_2, e_3, e_4\}$ is not independent.
- Goal: given E and π, find an independent subset of maximum weight.
- Strategy 1: from each component, select the largest element.
- Same as node strategy for head partition!

Maximum Weighted Matching

- Given a weighted graph, find a matching with the maximum total
- Simple greedy algorithm doesn't work!
- Simple algorithm will choose b,d, a,c, ignoring a,b, c,d

Matching

- Given: a weighted graph and a matching M on the graph
- Edges in M are **matched edges**
- Edges not in M are **free edges**
- Nodes not adjacent to matched edges are **exposed**
- Nodes that are adjacent to matched edges are **matched**

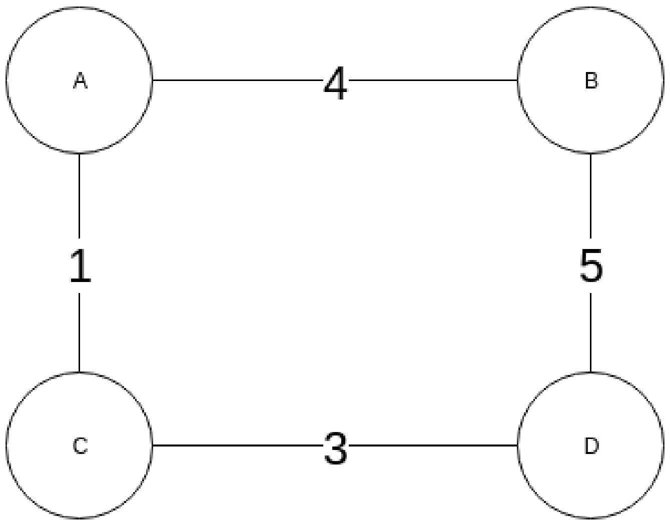


Figure 1: Weighted Matching Example

- **Augmenting Path**: a simple path in the graph beginning and ending at exposed nodes, and alternating between crossing matched and free edges.
- Augmenting Path example in 2: $(v_1, v_4, v_5, v_6, v_8, v_7, v_{10}, v_9)$
- Augmenting Path length always odd because start and end must be on exposed nodes.
- Finding longest augmenting path same as finding largest matching
- Given an augmenting path, swapping edges membership in M along the path creates a valid matching with length one longer.

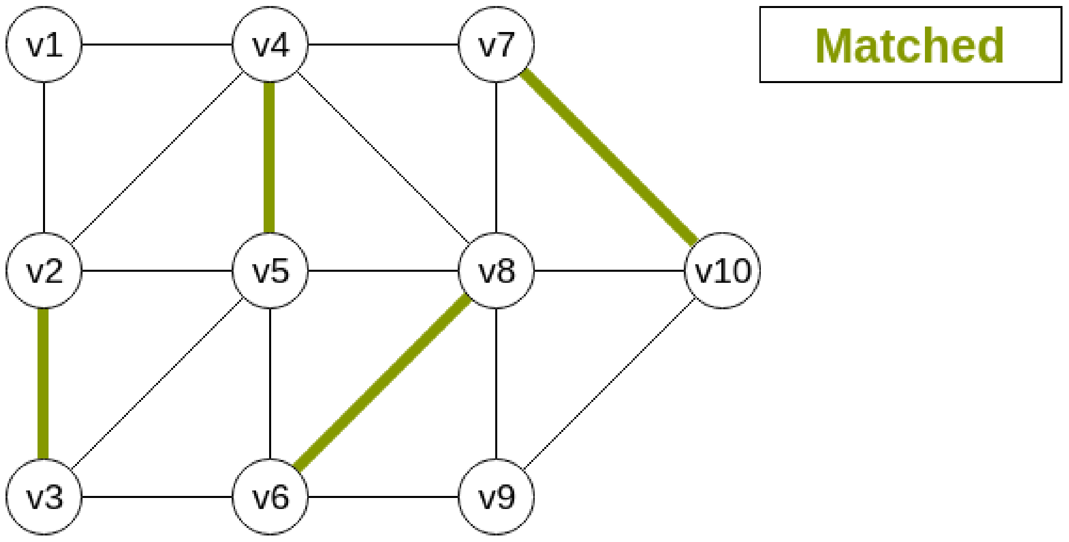


Figure 2: Matching Example

Day 8 Notes

Zach Neveu

May 16, 2019

1 Agenda

- Matching concepts
- Algorithm to solve matching
- Bipartite matching
- Intro to linear programming

2 Matching Review

- Review of matching problem from last class
- Swapping membership of augmenting path yields larger matching
- Maximum matching includes all nodes
- Key result: given a matching, M, the matching is optimal if and only if no augmenting path with respect to M exists.
- If no augmenting path → M is optimal (not so obvious)
- If M optimal → no augmenting path (this is fairly straightforward)

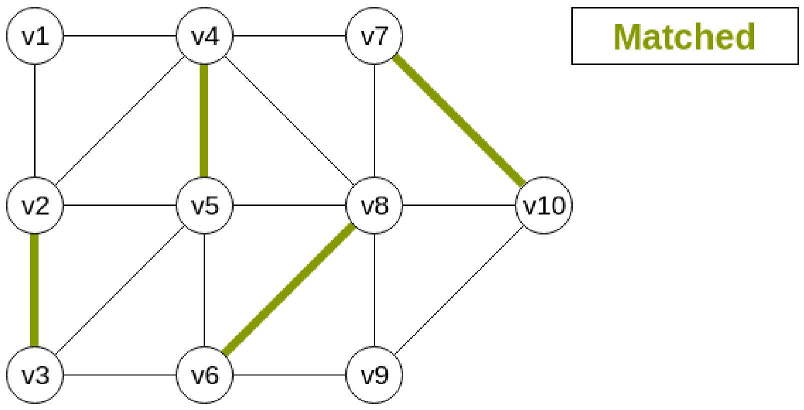


Figure 1: Review matching diagram from Day 6

Matching Algorithm

```
def match(g):
    m = 0
    while p = aug_path(M):
        swap_membership(p)
```

- Outer loop runs $O(E)$ times where E is edges
- Time to find an augmenting path is polynomial time
- Algorithm to find augmenting path is convoluted and not particularly useful to know

Bipartite Matching

- **Bipartite Graph**: a graph where the nodes can be divided into two groups such that every edge goes from one group to the other (no edges are inside a group).
- Bipartite Matching: find the largest matching in a bipartite graph
- Classic example problem: Job scheduling on heterogeneous computers
    - Group of jobs that all need to be done
    - Group of computers that can run jobs
    - Certain jobs can only run on certain computers
    - Find how to get the most jobs done

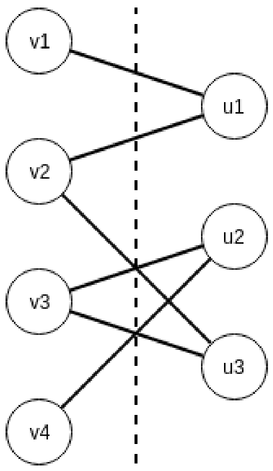


Figure 2: Bipartite Graph Example: Every edge crosses dividing line

- Start at all exposed nodes
- Do BFS across alternating edges until another exposed node is reached
- The path between exposed nodes that is found is an augmenting path
- BFS runs in fast polynomial time and fins any augmenting paths that exist

- Search diagram has structure: matched stages don't branch, matched and unmatched stages alternate
- Why is this like greedy algorithms?
  - Matching always getting bigger
  - Not quite greedy: some edges get deleted when swapping membership
  - Kind of a "deeper" greedy algorithm
  - Approach applicable to many problems

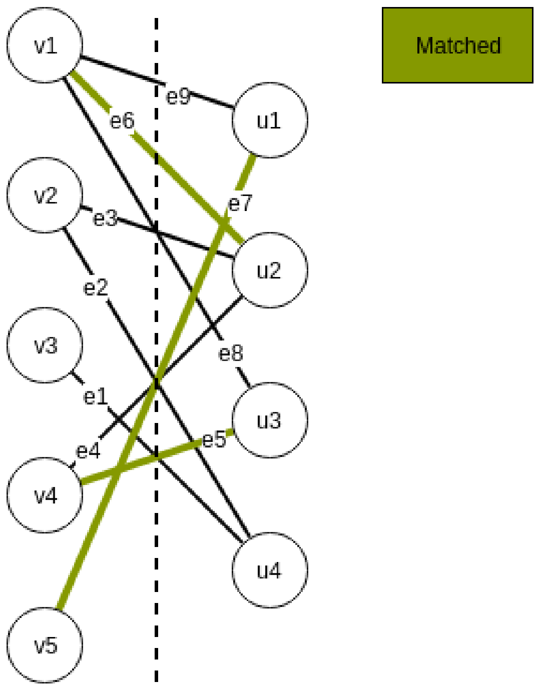


Figure 3: Exaple Bipartite matching

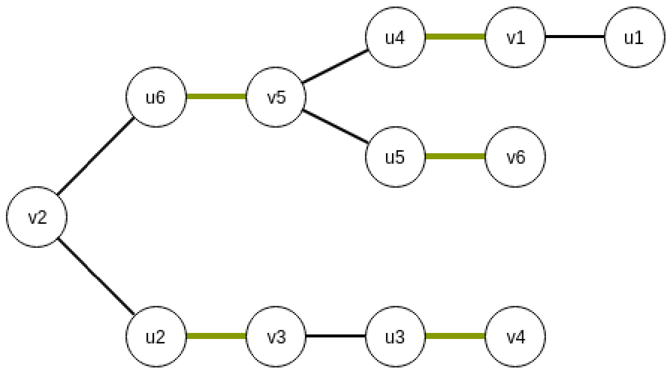


Figure 4: Example search diagram (different problem from fig. 3)

3 Linear Integer Programming (LP)

- Many problems have special format
- If your problem can be re-phrased into this format, it can be solved FAST by existing solvers.
- Leverage genius that is not yours
- Not super well known in ECE, came from applied math
- Large LP instances solvable in minutes
- Trade-off is that problem must be in exact format

Example

- You are a politician trying to win an election. Your district has 3 regions:
  - Urban - 100k voters
  - Suburban - 200k voters
  - Rural - 50k voters
- Goal is to get a majority in each region
- Win votes by advertising based on 4 issues
  - Building roads
  - Gun control
  - Farm subsidies
  - Gas taxes
- Goal: get max results within advertising budget

Table 1: Problem Breakdown Table

	urban	suburban	rural
Build roads	0.2	5	3
gun control	8	2	-5
farm subsidies	0	0	10
gas taxes	10	0	-2

Day 9 Notes

Zach Neveu

May 20, 2019

1 Agenda

- Quizzes back
- No simple mapping from # to grade - based on "expected number"
- Intro to LP
- Examples of LP

2 HW #2

- Definition of NP: **yes** example has certificate which can be verified in polynomial time.
- Definition is not symmetrical!
- Many problems in NP have inverse outside of NP

3 Linear Programming

- Example: Political Election categories (see day 8)
- Ex. Input: \$20k on roads, \$0 on guns, \$4k on farms, \$9k on gas
- Minimize $x_1 + x_2 + x_3 + x_4$ (total cost)

$$20(-2)+0(5)+4(0)+9(10) = 50{,}000$$
$$20(5)+0(2)+4(0)+9(0) = 100{,}000$$
$$20(3)+0(-5)+4(0)+9(-2) = 200{,}000$$

General Linear Program

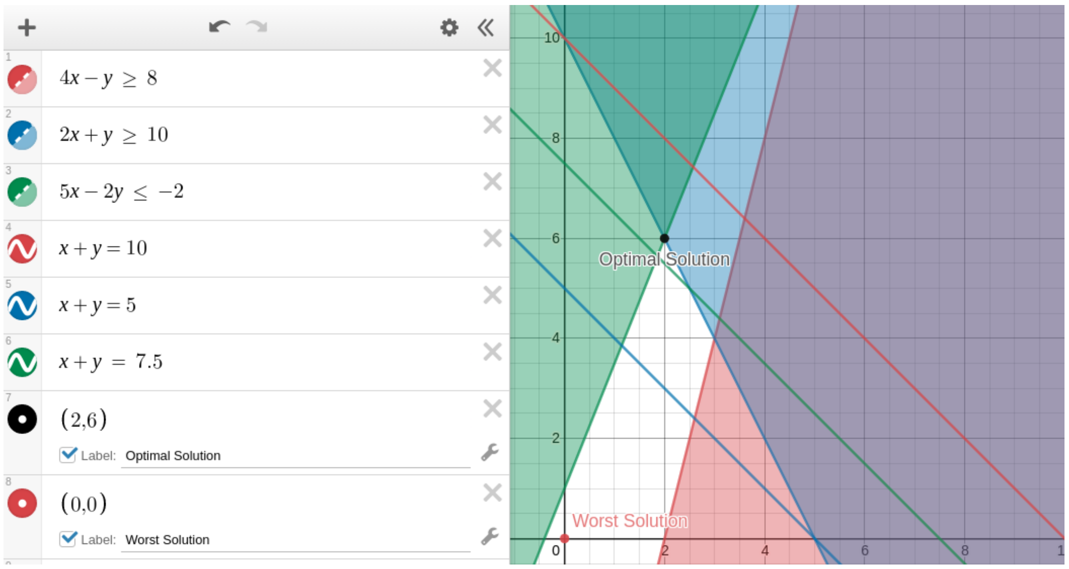
Given a set of constants $a_1, \ldots, a_n$ and a set of variables $x_1, \ldots, x_n$, a **linear function** f on the variables is:

$$f(x_1, \ldots, x_n) = a_1(x_1) + \ldots + a_n(x_n) = \sum_{\alpha=1}^{n} a_\alpha x_\alpha$$

Given a constant b and a linear function f,
$f() \leq b$ and $f() \geq b$ are linear inequalities
$f() = b$ is a linear equality

$<$ and $>$ are not linear!

LP Problem: The problem of maximizing or minimizing a linear function subject to a finite set of linear constraints.
Example:
Maximize: $x_1 + x_2$
Subject to:
$4x_1 - x_2 \leq 8$
$2x_1 + x_2 \leq 10$
$5x_1 - 2x_2 \geq -2$
$x_1, x_2 \geq 0$



$4x - y \geq 8$
$2x + y \geq 10$
$5x - 2y \leq -2$
$x + y = 10$
$x + y = 5$
$x + y = 7.5$
$(2,6)$ Label: Optimal Solution
$(0,0)$ Label: Worst Solution

Figure 1: Graph of Solution Space: where all regions overlap is valid (marked in white). $x = y$ lines show value -> further from origin is better.

Example: Reclaiming Solid Waste

Inputs are 3 types of materials: 1,2&3

Table 1: Material Availability

Material	Available Pounds/Week
1	100
2	200
3	300

Problem:
Maximize: $5Y_A + 10Y_B$

Table 2: Grades

Grade	Spec	profit/pound
A	$M_1 \leq 30\%, M_2 \leq 40\%$	5
B	$M_2 = 50\%, M_2 \leq 20\%$	10

Let: Z_{MN} = the proportion of grade M that is material N
Subject to:
$Z_{A_1} \leq 0.3$
$Z_{A_2} \leq 0.4$
$Z_{B_2} = 0.5$
$Z_{B_3} \leq 0.2$
$Y_A * Z_{A_1} + Y_B * Z_{B_1} \leq 100$
$\vdots$
UH OH! This multiplies variables which isn't linear...
Fix:
Let: X_{MN} = <u>amount</u> of Material N in grade M
New Constraints:
$X_{A_1} + X_{B_1} \leq 100$
$X_{A_2} + X_{B_2} \leq 200$
$X_{A_3} + X_{B_3} \leq 300$
New Objective Function:
Maximize: $5(X_{A_1} + X_{A_2} + X_{A_3}) + 10(X_{B_1} + X_{B_2} + X_{B_3})$

Takeaways

- Variable choices matter! Effects speed and solve-ability
- Seemingly non-linear variables can be re-written as linear variables

Standard Form

Given constants $c_1, \ldots, c_n$, $b_1, \ldots, b_m$ and $m * n$ values a_g for $i = 1 \to m$ and $j = 1 \to n$, find $x_1, \ldots, x_n$ such that:
Maximize: $\sum_{j=1}^{n} c_j x_j$
Subject to: $\sum_{i=1}^{n} a_{ij} x_j \leq b_i$ for $i = 1 \to m$
$x_j \geq 0$ for $j = 1 \to n$
n = # variables m = # constraints

- In standard form, all variables $x \geq 0$
- Only maximization as operation
-

Concise Standard Form

- $\boldsymbol{A} = (a_{ij})$
- $\boldsymbol{b} = (b_i)$
- $\boldsymbol{c} = (c_j)$
- $\boldsymbol{x} = (x_j)$

An LP formulation in standard form is: maximize $C^T x$ subject to $Ax \leq b$, $x \geq 0$.

Day 10 Notes

Zach Neveu

May 21, 2019

1 Agenda

- Quiz
- LP Standard Form
- More LP examples
- Solving LP Problems

2 LP Examples

Personnel Scheduling Problem

- Goal: find an assignment of people to shifts such that enough people are working in each slot and total cost is minimized.
- Objective function: $z = 170x_1 + 160x_2 + 175x_3 + 180x_4 + 195x_5$
- Constraint example: $x_1 + x_2 \geq 79$
- Constraint form: $\sum$ active shifts during period $\geq$ required personnel during period.
- Following constraints for all time periods create many redundant constraints
- **redundant constraint:** Constraint can be deleted without changing the solution to the LP
- Solution to this problem: x=(48,31,39,43,15), z=30,610

Table 1: Personnel Schedules

Shift	Time Range	Hourly Wage $
1	6a-2p	170
2	8a-4p	160
3	12p-8p	175
4	4p-12a	180
5	10p-6a	195

3 LP Standard Form

- Maximize $\sum_{j=1}^{n} c_j x_j$ subject to $\sum_{j=1}^{n} a_{ij} x_j \leq b_i, i = 1 : m, x_j \geq 0$

Table 2: People Required

Shift	People Needed
6-8	48
8-10	79
10-12	65
12-2	87
2-4	64
4-6	73
6-8	82
8-10	43
10-12	52
12-6a	15

- Always maximizing
- Constraints always use $\leq$
- All variables ≥ 0
- Any LP problem can be rephrased into a standard form equivalent
- **Equivalent:** Two LP formulations are equivalent if they are both maximizing and for every feasible solution in L there is a corresponding feasible solution in L' with the same objective value and visa versa.
- If L is minimizing and L' is maximizing, multiply objective function by -1
- If not all variables are constrained by $\geq 0:$ create 2 positive variables and take their difference to get the unbounded value (e.g. $x_2 \rightarrow (x_2' - x_2''))$
- If equality used instead of inequality in constraint: Split into $\leq$ and $\geq$ constraints, and flip $\geq$ one.

Day 11 Notes

Zach Neveu

May 22, 2019

1 Agenda

- LP solving with simplex
- Introduction to LP
- Homework #3 due Friday
- Quiz #3 next Tuesday

2 LP

Options for LP soln:

- Single possible solution
- No feasible solution
- Unbounded feasible region
- Infinite optimal solutions along line segment

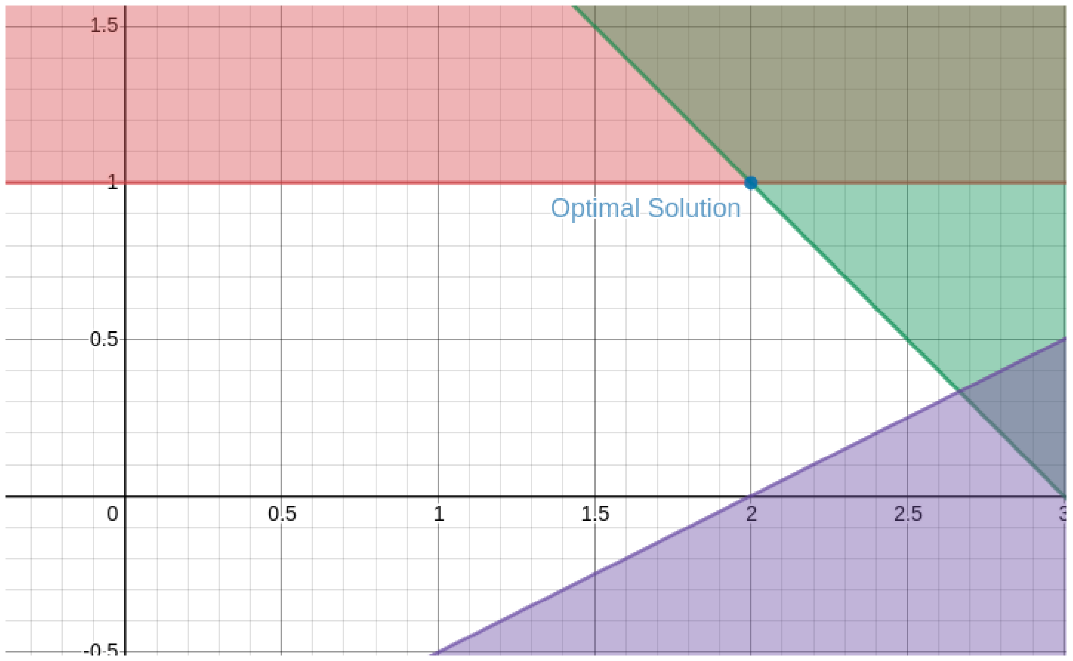


Figure 1: Single Solution

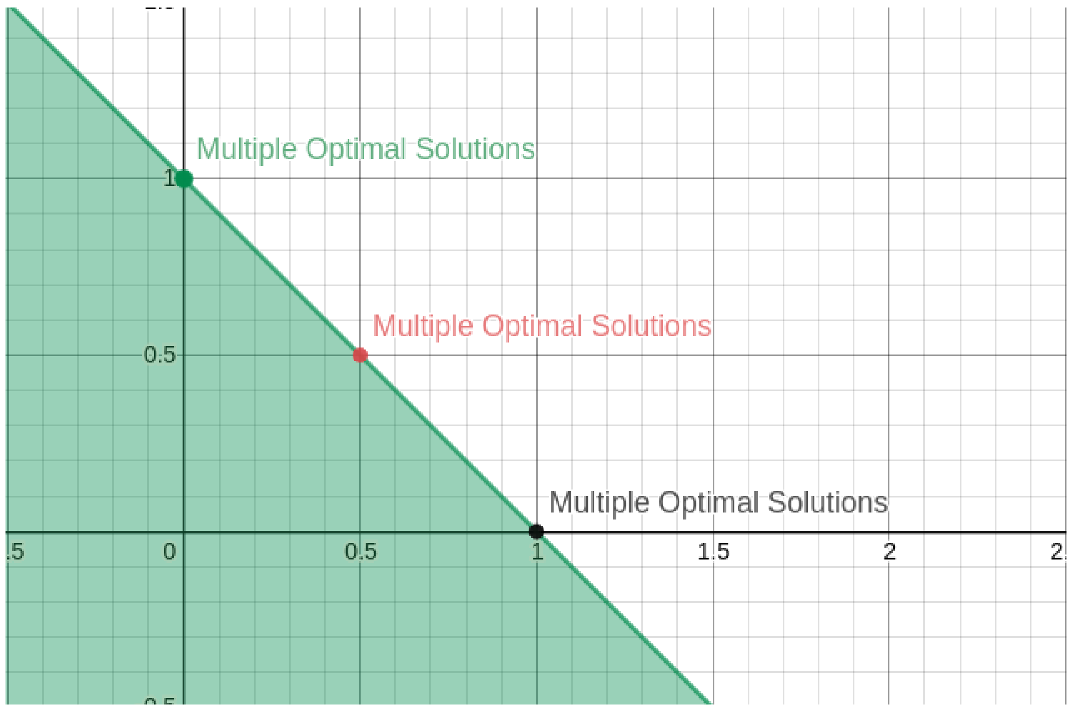


Figure 2: Multiple Solutions Along Segment

- **Cornerpoint Feasible Solution (CPF Solution):** Cornerpoint solutions that border the feasible region.
- Number of cornerpoints $\leq \binom{constraints}{2}$
- Adjacent CPF solutions are connected along the edge of a constraint
- Optimality Test: If a CPF Solution is better than all adjacent CPF solutions, then it must be optimal.
- This works because simplex is always **convex**
- **Convex:** Lines connecting all pairs of nodes fall completely within the enclosed area
- **Boundary** of the feasible region is the parts of the constraint boundaries that are feasible
- CPF solutions in 3 dimensions are the intersection of 3 planes
- In 3 dimensions, each CPF solution has up to 3 possible adjacent solutions
- **Edge:** The feasible portion of the intersection of 2 constraint boundaries
- Number of possible CPF solutions can be exponentially large (num constraints choose num dimensions)
- Not trivial to search through all possible CPF solutions in certain cases
- Bad news: Possible to create an intractable simplex instance
- Good news: In practice, instances are solved very fast
- More Good news: other algorithms can solve LP problems in P!

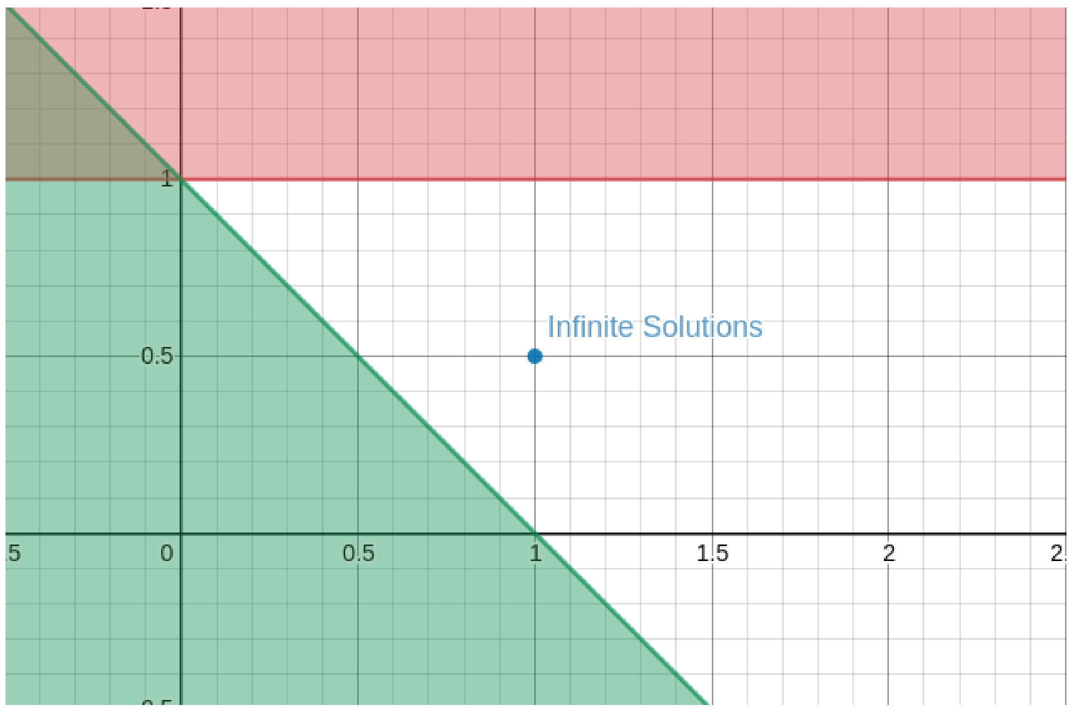


Figure 3: Unbounded Feasible Region

Simplex Algorithm

1. Initialization: pick an initial CPF Solution
2. Optimality test
3. Select new CPF Solution that is adjacent
4. Go back to 2 until optimality test passes.

3 Using LP in Class

AMPL Notes

```
# Write in *.mod file
var x1;
var x2;
maximize objective: x1+x2;
subject to constraint1:
    4*x1-x2 <= 18;
constraint2:
    2*x1+x2 <=  18;
```

1. Write AMPL commands into *.mod file
2. Run `ampl` at unix prompt to launch AMPL

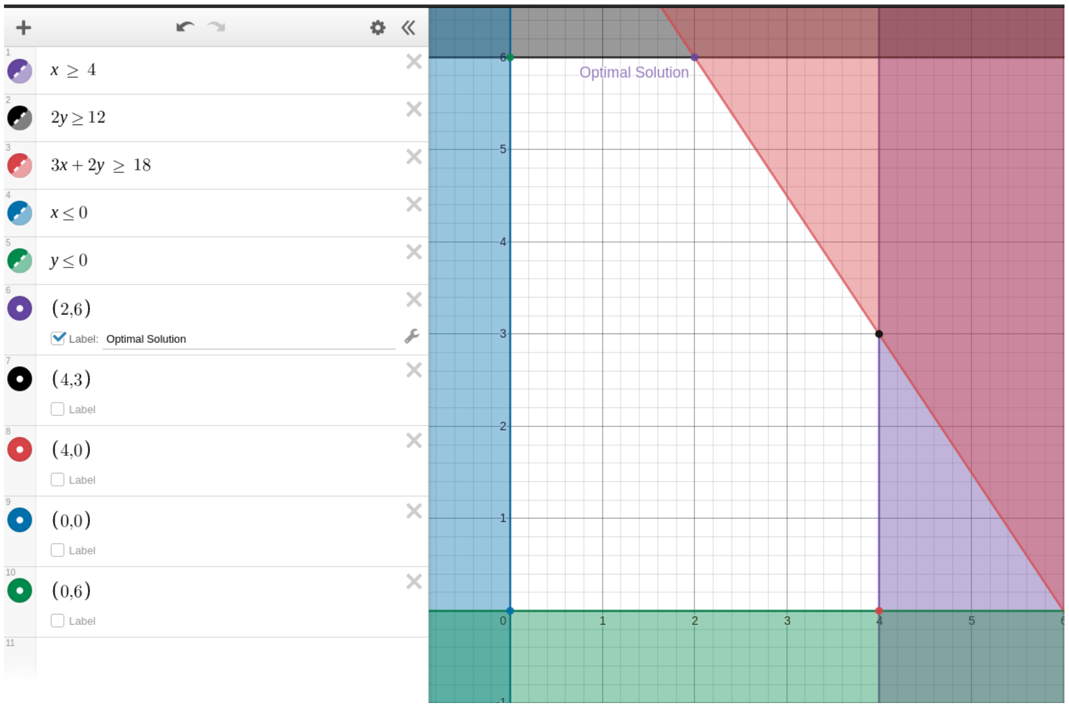


Figure 4: Example LP from Text Book

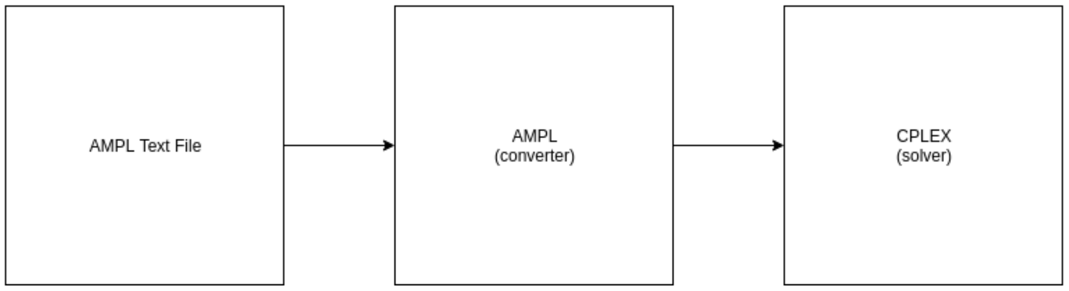


Figure 5: Flow Diagram of Solving Process

3. run `model *.mod` to load model
4. run `data *.dat` to load data
5. run `solve;` to solve
6. Solver spits out # of simplex steps required
7. run `display x1;` to display result
8. run `include *.run` to run script of AMPL commands
9. run `ampl *.run` from unix prompt to run script

AMPL Examples

```
param N=10;
var x {i in 0..N-1};
subject to C1:
    x[0]+3*x[i] <=  10;
c2:
    sum{i in 0..N-1} x[0] = 0;
c3 {j in 0..10}: x[j] = x[j+1];
```

- Variables $\neq$ Parameters
- Parameters are known at solve time - unchanging
- Variables: may change in execution
- Store params in data file
- Store variables in model file

Day 13 Notes

Zach Neveu

May 28, 2019

1 Agenda

- ILP Advanced Techniques
- Project 3
- Intro to Branch & Bound
- TODO:
  - Reading
  - HW due Tomorrow
  - Project due Monday
  - Quiz next Tuesday

2 ILP

$3\text{x}_1 + 2x_2 \leq 18$
OR
$x_1 + 4x_2 \leq 16$

- Binary variables x_1, x_2
- How can we phrase this in terms of ILP?
- Typically, both would need to be solved
- Add a large number multiplied by a new binary variable
- Negate new variable in one equation so that this variable can control
- Now at least one constraint must be satisfied, or both can be satisfied

$3\text{x}_1 + 2x_2 \leq 18 + yM, M \to \infty$
$x_1 + 4x_2 \leq 16 + (1 - y)M$

- Say we have three constraints now
- How to implement or with more than 3 inputs?
- Add 3 new variables
- Add constraint that 3 variables sum to less than 3

$3\mathrm{x}_1 + 2x_2 \leq 18 + y_1 M$
$x_1 + 4x_2 \leq 16 + y_2 M$
$2x_1 + 3x_2 \leq 14 + y_3 M$
$y_1 + y_2 + y_3 \leq 2$

3 Branch & Bound

- Useful for solving ILP
- Also very useful for other stuff
- Based on divide and conquer
- In 1, $x_0 = (1.5, 2.5)$ is the solution to the LP relaxation of this problem.
- x_0 is non-integral, so we keep going
- Subdivide the feasible region into two non-overlapping feasible regions such that no feasible solutions are excluded.
- SP1: add constraint $x \leq 1$
- SP2: add constraint $x \geq 2$
- Keep subdividing on integer boundaries
- for SP1: add $y \leq 1$ SP3, add $y \geq 2$ SP4
- For SP2: add ≤ 1 SP5, $y \geq 2$ SP6
- SP4, SP6 infeasible, no points in those areas
- Find LP solns to SP3, SP5
- LP soln to SP3 turns out to be integral - no need to keep going
- For SP5, add $x \leq 2$ SP7, and $x \geq 3$ SP8
- SP8 infeasible
- SP7 has integral soln
- Compare integral solutions to SP7 and SP3
- No subproblems left to expand, choose best LP soln (SP7)

4 Project

- Solve knapsack and coloring using ILP
- Instances all converted already

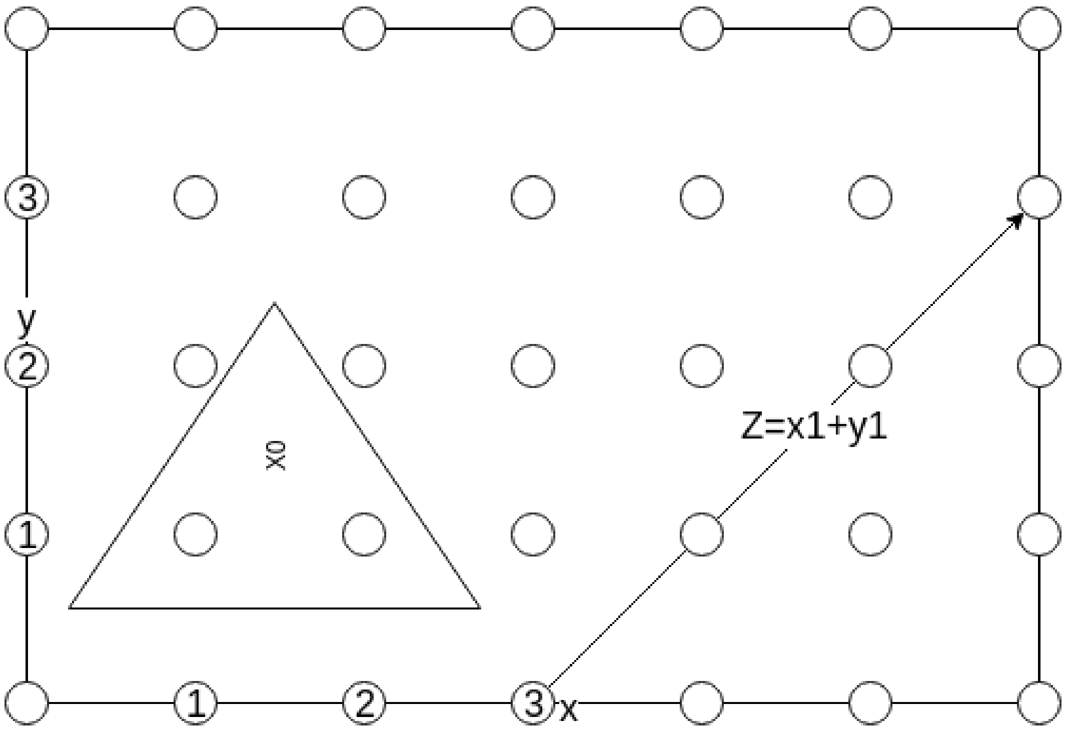


Figure 1: ILP w/ Branch and Bound Example

- Must solve formulation of problem
- Turn in model files, output files
- Limit solver runtime to 10 minutes

Day 14 Notes

Zach Neveu

May 29, 2019

1 Agenda

- Branch and Bound to solve ILP
- Announcements
  - Reading
  - Project due Monday
  - Quiz Tuesday

2 Branch & Bound

- Review of last class example
- LP soln is **optimistic bound** on ILP soln
- Add constraints to break up problem into subproblems
- Solve each subproblem with LP solver
- Repeat recursively until subproblem has integral soln in LP
- Don't expand subproblem if...
  - Subproblem is infeasible (No LP solns)
  - LP soln in integral
- How to explore graph?
- Best soln is to use DFS
- Z score of LP soln at parent node is upper bound on Z score of children
- This means that branches with low Z scores can be pruned!
- Suddenly problem not exponential time any more!

Fathoming Rules: don't expand a subproblem if...

1. Subproblem is infeasible
2. LP solution is integral

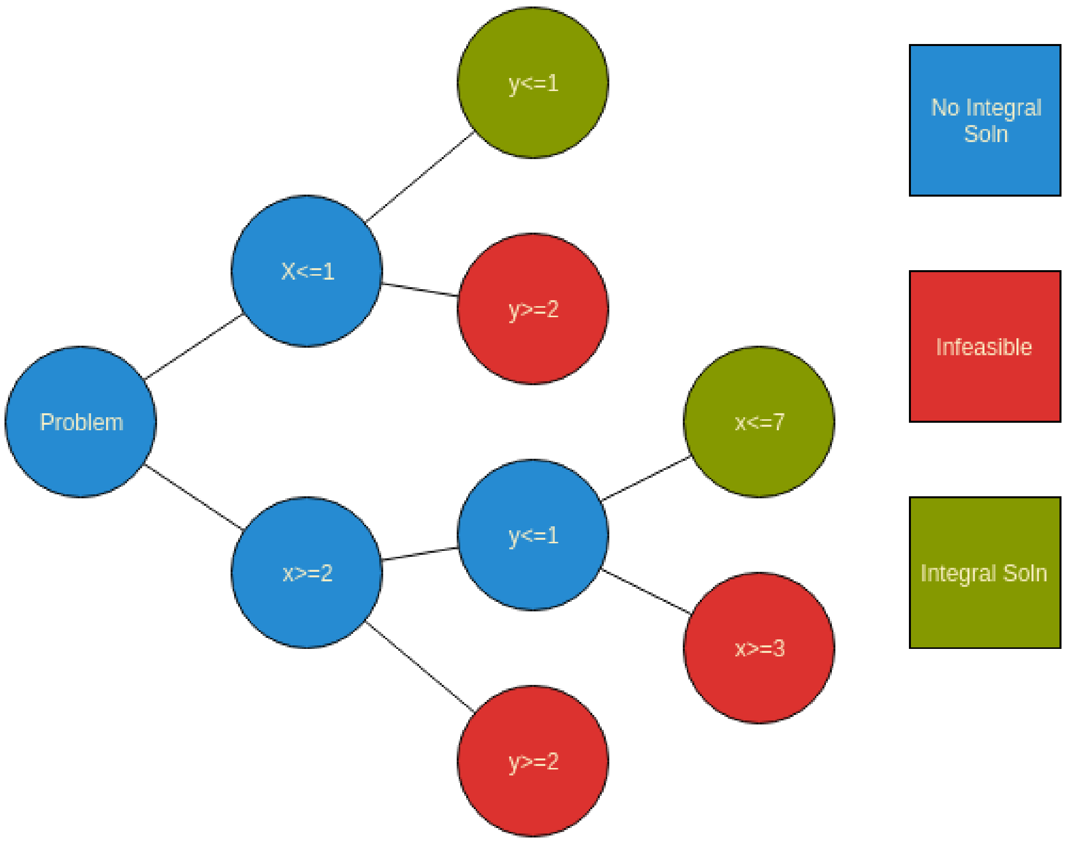


Figure 1: Subproblem Search Diagram

3. Optimistic bound is worse than the best integral solution found so far

- This only prunes if the bound is tight.
- Branches with loose bound will not be pruned until far down the tree
- LP bound is quite tight
- What order to branch variables? x first? y first?
- Crucial decision, but no fixed best algorithm - solvers offer choice
- **IDEA:** Try different sized problems with different strategies, find optimal domain for each strategy. Potentially weed out strategies that don't have a region where they are optimal.

Problem Example

$$
\begin{gathered}
Z = 9x_1 + 5x_2 + 6x_3 + 4x_4 \\
6x_1 + 3x_3 + 5x_3 + 2x_4 \le 10 \\
x_3 + x_4 \le 1 \\
-x_1 + x_3 \le 0 \\
-x_3 + x_4 \le 0 \\
x_i \in \{0,1\}
\end{gathered}
$$

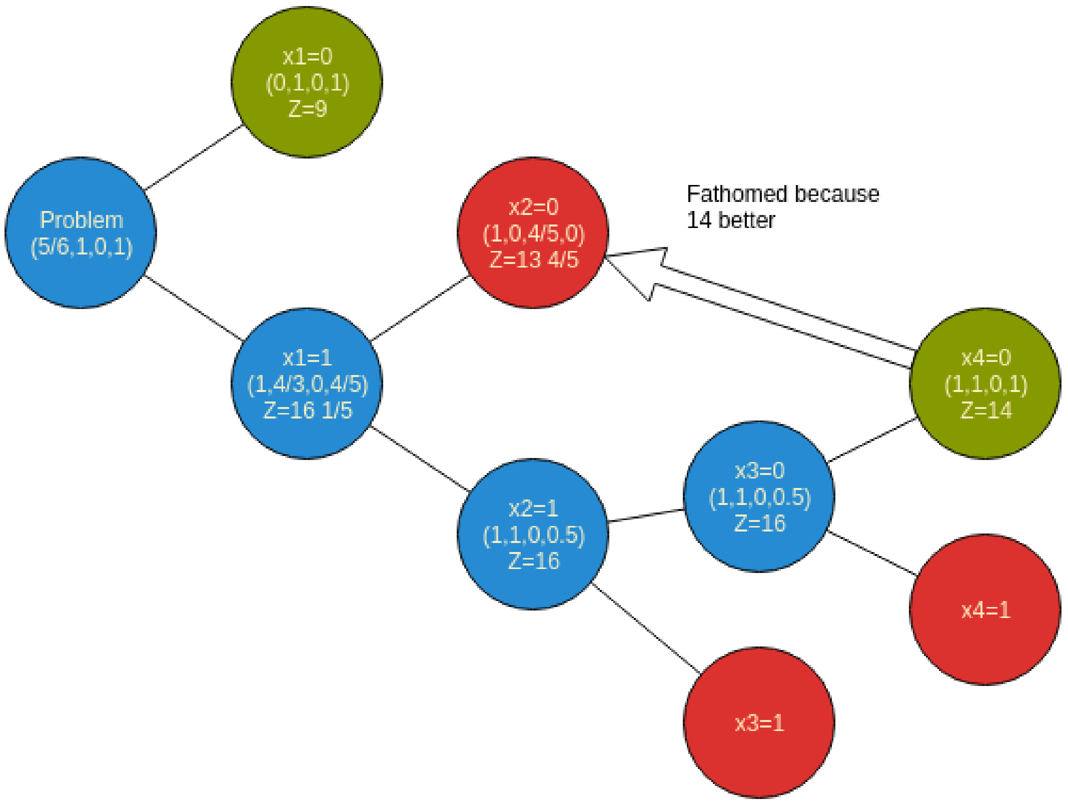


Figure 2: Example Problem Search

3 Project Help

Graph Coloring ILP

- Goal: Minimize Conflicts
- Constraints: # colors, edges
- Graph given as list of edges -> constant set param in ampl

- Convert into adjacency matrix -> Boolean matrix (numNodes x numNodes)
- Possible variables:
  - `Colors[numNodes] = {0..numColors}` -> hard to create linear objective
  - `Colors[numNodes,numColors] = {0,1}` -> Conflicts easy! Just sum of columns!

Table 1: caption

	c1	c2	c3	c4
node 1	0	1	0	0
node 2	0	1	0	0
node 3	0	0	1	0
conflicts	0	1	0	0

```
param edge {i in 0..numNodes-1, j in 0..numNodes-1} :=
    (if(i,j) in edgelist or (j,i) in edgelist then 1 else 0);
```

4 Advanced ILP Techniques

Fix Variables

- Consider constraint $3x_1 \leq 2$. x_1 must be 0.
- $3x_1 + x_2 \leq 2$: x_1 must be 0.
- $5x_1 + x_2 - 2x_3 \leq 2$: x_1 must be 0.
- In general, constants bigger than constraint must be set to 0.

Eliminate Redundant Constraints

- If constraint depends entirely upon value of different constraint, eliminate it.

$$\begin{aligned} 3x_3 - x_5 + x_7 &\leq 1 \\ x_2 + x_4 + x_0 &\leq 1 \\ x_1 - 2x_5 + 2x_6 &\geq 2 \\ x_1 + x_2 - x_4 &\leq 0 \end{aligned}$$

- Fix variables
  - $x_3 = 0$
  - $x_6 = 1$

- $x_5 = 0$
- $x_2 = 0$
- $x_1 = 0$
- x_7 is only real free variable!

Cutting Planes

- Consider problem in fig. 3
- Top right corner points are not integral solutions
- Can eliminate these infeasible points before sending to solver
- Do this elimination by adding a redundant constraint that all integral feasible solutions satisfy
- Consider $6x_1 + 3x_2 + 5x_3 + 2x_4 \leq 10$
- If $x_1, x_2, x_4 = 1$, then this doesn't work
- Add $x_1 + x_2 + x_4 \leq 2$ to speed up ILP solver.
- Without redundant constraint, solver produces $(\frac{5}{6}, 1, 0, 1)$, but the redundant constraint does not allow for this, allowing faster fathoming.

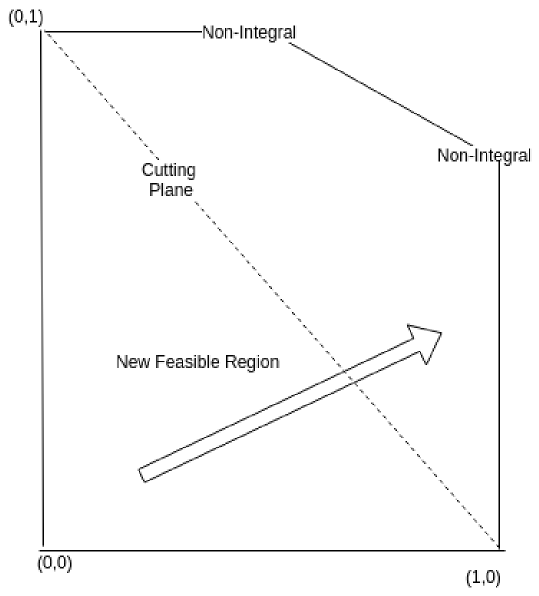


Figure 3: Cutting Plane Example

Day 15 Notes

Zach Neveu

May 30, 2019

1 Agenda

- Wrap up advanced ILP techniques
- Applying branch and bound

2 Advanced LP Techniques

- Review of cutting plane example from day 14.

3 Branch and Bound Beyond ILP

- Job scheduling problem
- Given a set of jobs to process.
- Each job takes the same amount of time to process.
- Setup time depends on the current job and the previous job
- Goal: minimize total time - same as minimizing setup time
- Brute force - $O(n!)$

Table 1: Job Scheduling Example Problem

Type	1	2	3	4	5
None	4	5	8	9	4
1	0	7	12	10	9
2	6	0	10	14	11
3	10	11	0	12	10
4	7	8	15	0	7
5	12	9	8	16	0

- Branch and Bound technique
- Possible starting bounds
  - $-\infty$
  - 0

  - min value * num jobs - 20 in this case
  - $\sum$ min value in each column - 30 in this case
  - Can't easily tighten this now, but as tree progresses can tighten quickly
- Assume schedule begins with (3,1) - cost so far $8 + 10 = 18$
- To compute bound, eliminate rows `none` and 3
- Take minimum from remaining problems excepting these rows
- Optimistic bound $8 + 10 + 7 + 10 + 7 = 42$
- This bound is infeasible solution - jobs 2 and 4 both done directly after job 1
- This seems similar to the infeasible optimistic bounds from solving ILP using LP!
- If bound is feasible, then this path is fathomed!

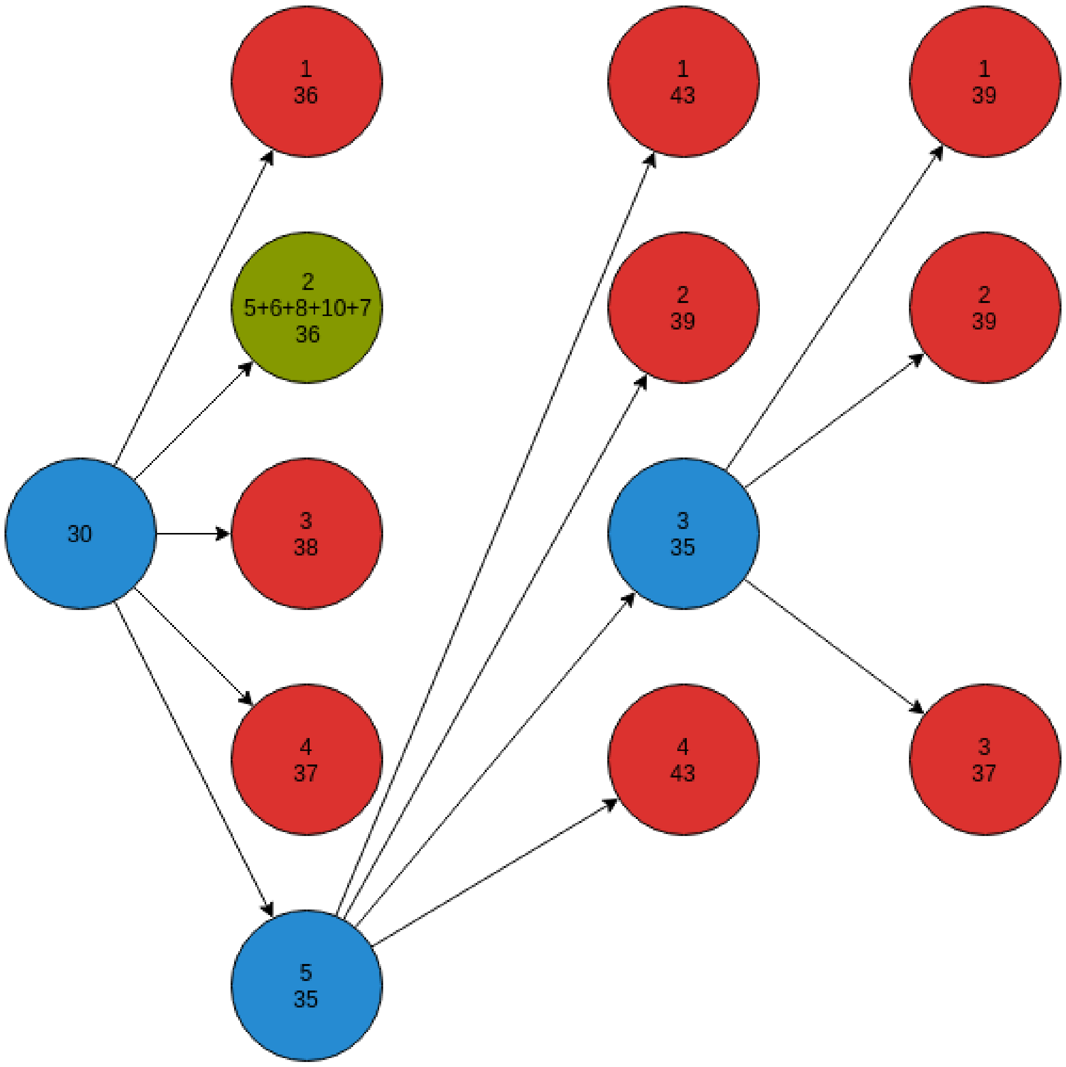


Figure 1: Job Scheduling Search Tree

Knapsack w/ Branch & Bound

Table 2: Knapsack Instance, cost limit = 100

	1	2	3	4	5	6
v_i	10	35	40	18	2	4
w_i	10	50	100	45	5	20
ratio	1	0.7	0.4	0.4	0.2	

- Sort by value/weight ratio
- Start tree at highest ratio item
- Optimistic bound:

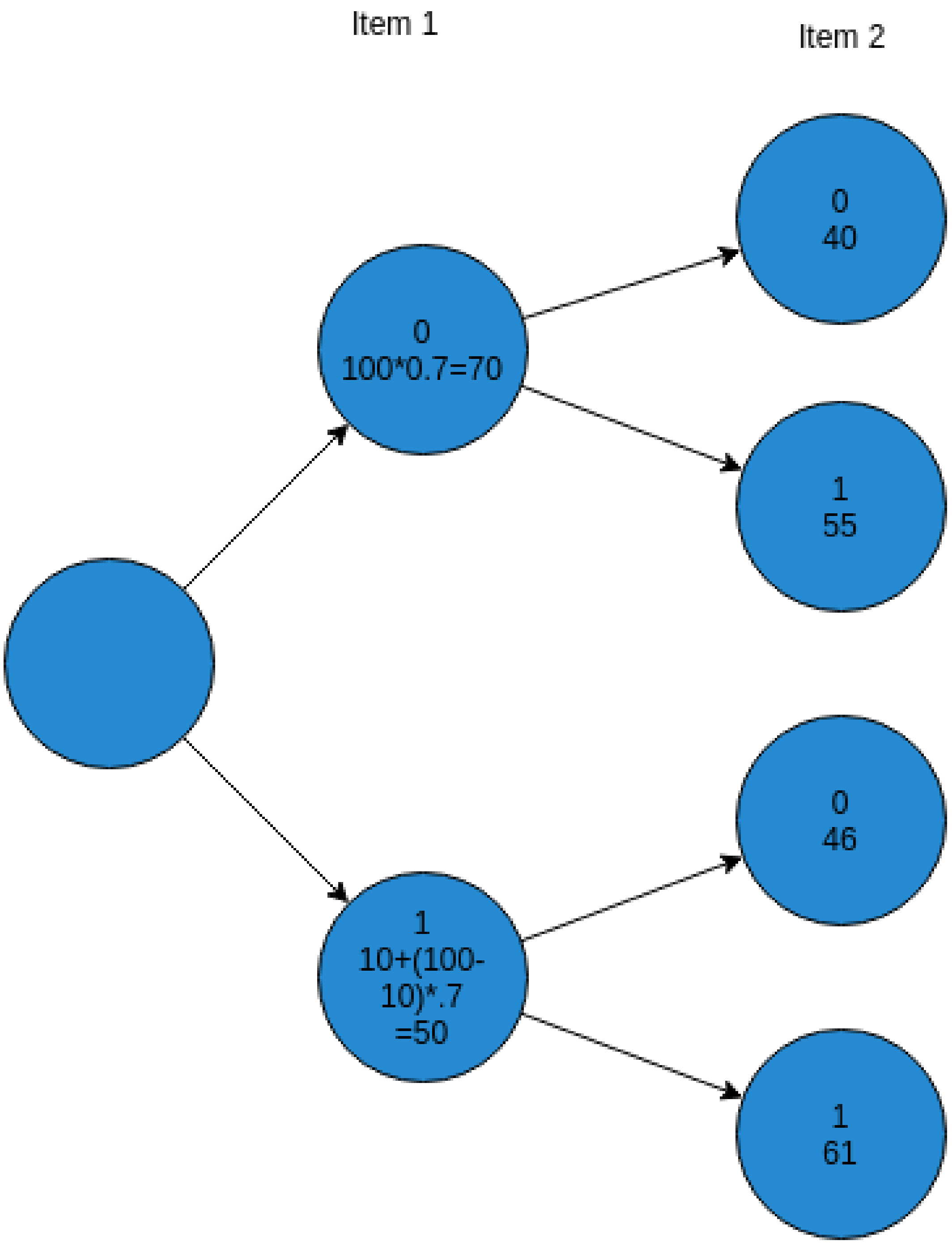


Figure 2: Knapsack Example

Day 16 Notes

Zach Neveu

June 3, 2019

1 Agenda

- Knapsack branch and bound
- Project #4 Introduction
- Intro to Dynamic Programming
- Quiz Tomorrow
- HW due Thursday 6/6
- Project 4 due Monday 6/10
- Quiz Feedback

2 Knapsack Branch and Bound

- Recall example from last class
- Key Ideas
  - Branch on each item in knapsack
  - Estimate bound by filling remaining space with ratio of next highest item
  - This bound is somewhat loose - obviously not all items can be this good
- Improved bound: Add available items, when item won't fit, include the largest fraction of it that will!
- This is just like the LP bound on ILP!
- This knapsack problem, where items can be partially included is called **Fractional Knapsack**
- Solving fractional knapsack gives tight bound on solving 0-1 knapsack.

3 Project 4 Introduction

- 2 Part project
- Given header file for knapsack object
- implement `Knapsack::bound()` which finds the improved bound above

  - Beware special cases!

- Implement `Knapsack::branchandbound()` searches for optimal solution using `bound()`
  - Decide how to branch on variables
  - Decide order to divide subproblems
  - Use knapsack object to store subproblems - each subproblem gets a knapsack
  - Use variable `num` to indicate how many variables have been fixed
  - Suggestion: use list structure (stl::deque) of subproblems. Choose item from deque, expand it, add new subproblems back to deque.
  - When feasible solns found keep track of the best one
  - 10 Minute limit

4 Dynamic Programming

- **Mysterious**
- NOT related to dynamic memory stuff
- Advanced algorithm design: make sure you are doing everything exactly once, not less, not more.
- **Dynamic Programming** is about not duplicating work
- Requires very specific properties. Rarely useful, but very good when it works.
- Revolves around Multiplying N matrices {a1, a2, a3, ..., aN}
- What order do multiplications go? For N=3, can use (A1*A2)A3 or A1(A2*A3).
- Order doesn't effect result, but does effect work required!
- Assume multiplying pxq and qxr matrices to get pxr.
- Final result is computing $p*r$ values
- Multiplications generally much slower than additions
- q products required to compute each entry of result
- pqr total products required - n^3 time, kinda slow

$\mathrm{A}_1 \rightarrow 10x100$
$A_2 \rightarrow 100x5$
$A_3 \rightarrow 5x50$
$(A_1A_2)A_3 \rightarrow (10*100*5)+(10*5*50) = 7500$
$A_1(A_2A_3) \rightarrow (100*5*50)+(10*100*50) = 75,000$

- One answer is 100x harder to get!!

- How to minimize the work done?
- Number of ways to compute - huge for large N
- **Parenthesization** - number of ways to add parentheses to group values

$\mathrm{A}_1, A_2, A_3, A_4, \ldots, A_{n-1}, A_n$
$p_0 x p_1, p_1 x p_2, p_2 x p_3, \ldots, p_{n-1} x p_n$
$A_{i..j} = A_i A_{i+1} A_j \rightarrow PartialProduct$

- Assume an optimal parenthesization of some product has been found
  1. There must be a final product to compute
  2. Total cost of that optimal parenthesization = (cost to compute final product + price to compute LHS + price to compute RHS)
  3. **Optimal Substructure Property**: Optimal parenthesization of entire problem must contain optimal parenthesizations of subproblems. If LHS+RHS+FP is optimal, then LHS & RHS must both be optimal as well.
- Notation
  - Let $m[i, j]$ = Minimum # of multiplications needed to compute subproduct $A_{i..j}$
  - Looking for $m[1, n]$ eventually
  - Final multiplication: $A_{i..k} * A_{k+1..j}$.
  - $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$
  - Problem: optimal k unknown...
  - Try all values of k and pick the best! Only $O(n)$
  - $m[i, j] = 0 if(i == j)$, else $argmin_k(m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$

Day 17 Notes

Zach Neveu

June 4, 2019

1 Agenda

- Quiz
- Review of Dynamic Programming

2 Dynamic Programming

- Problem: order of matrix multiplication
- Key concepts
  - All solutions must have a final multiplication
  - Final multiplication cost is LHS+RHS+combining
  - For optimal soln, LHS, RHS must each be optimal
  - This is true recursively
- Notation
  - $m[i,j]$ - optimal sub-product cost $\prod_{k=i}^{j} A_k$
  - $m[i,j] = 0$ if $i == j$, else $argmin_k(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j)$

```python
# Recursive Matrix Chain
# Computes m[i,j]
def RMC(p,i,j):
    if i == j:
        return 0
    else:
        m[i,j] = MAXINT
        for k in range(i,j-1):
            q = RMC(p,i,k)+RMC(p,k+1,j)+p[i-1]*p[k]*p[j]
            if q < m[i,j]:
                m[i,j] = q
    return m[i,j]
```

- Quiz question: modify this algorithm to keep track of best k at each step!
- Problem: Same sub-problems are solved over-and-over
- Just like recursive Fibonacci

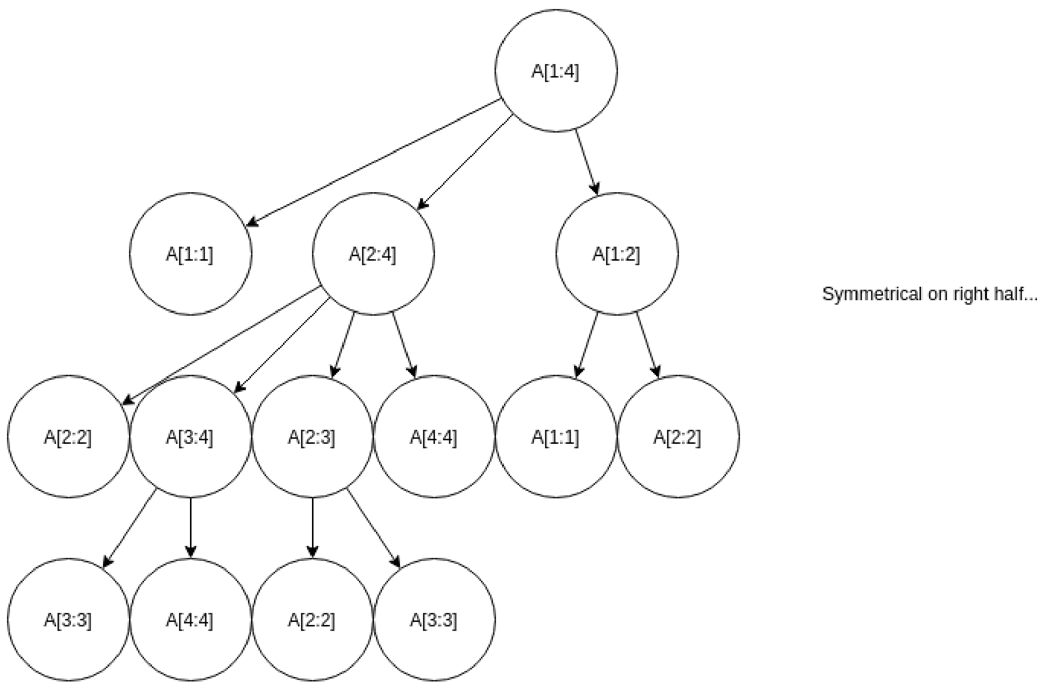


Figure 1: Example RMC solution tree for $A_{1..4}$

- Solution: make $m[]$ global, and check if soln to subproblem has already been found. Use answer if it has.
- **memoization**: This process of saving already-computed solutions

Complexity

- m matrix has n^2 entries
- Each entry computed exactly once
- Work to compute each entry (or at least one half (diagonal symmetry)):
  - Each func call takes $O(n)$
  - Total function takes $O(n * n^2) = O(n^3)$

Optimization

- Idea: instead of discovering tree recursively, why not just go through matrix in order and solve each subproblem?
- Organize as triangle (half matrix) w/ i,j along edges
- Build values from trivial diagonal to corner
- **dynamic programming** technically refers to this triangle approach, not the memoization

Dynamic TSP Example

- Is set of consecutive indices in HC also a HC? NO!
- Instead, given a subset of nodes, s, and some $k \in s$, $c(s,k)$ = min cost to start at node 1, visit all nodes in S, and end at k.
- $c()$ can be expressed in terms of subproblems easily.
- Remove k^{th} node from the set and repeat with new k
- Try all k and pick optimal
- Example
  - c({2,4,6,7}, 2) is min(c({4,6,7},4)+w[4,2], c({4,6,7}, 6)+w[6,2], . . .)
  - In general, $c(S,k) = min_{m \in S-\{k\}} c(s-\{k\},m) + w[m,k]$

Day 18 Notes

Zach Neveu

June 5, 2019

1 Agenda

- Dynamic TSP Review
- Local Search

2 Dynamic TSP

- Review concept of Hamiltonian paths $c(\boldsymbol{S}, k)$
- To optimize $c(\boldsymbol{S}, k)$, try all possible last nodes on the path before k, and choose minimum.
- $c(\boldsymbol{S}, k) = argmin_k[c(s - \{k\}, m) + w[m, k]]$
- $c(\{2\}, 2 = w[1, 2]$ base case. Cheapest HP with one node is just weight of single path.
- $c(\{4, 2\}, 2) = c(\{4\}, 4) + w[4, 2]$
- $c(\{2, 4, 6\}, 2) = min_k(c(\{4, 6\}, 4) + w[4, 2], c(\{4, 6\}, 6) + w[6, 2])$
- Once all costs are generated from 1 to all other nodes, just add paths from last node back to 1 and minimize.
- Problem: $O(n_2^n)$ entries, each computation is $O(n)$, so complexity is $O(n^2 2^n)$
- Exhaustive algorithm requires $O(n!)$
- 2^n smaller than $n!$, so we actually have a good improvement!
- Difference is, exhaustive algorithm re-solves all subproblems every time.
- **quiz q idea**: write top down/bottom up algorithm to solve TSP this way

3 Local Search

- "Blunt tool" that works almost everywhere
- Usually a heuristic, not exact
- Really dang good results sometimes
- Key idea: Good solutions lie in similar areas. If you have a good solution, try tweaking it a little to see if it gets better.
- Consider optimization problems in form (F, c) where F is feasible set and c is cost function

- Given a **neighborhood function** which maps each element of F onto a subset $f \in F$
- Given F, $N(t)$, is set of feasible points "near" t
- **Improvement function**: `improve(t)` returns any neighbor with lower cost, or none if none exist.

```
# Generic Local Search Algorithm
t = rand(F) # random point in F
while improve(t) is not None:
    t = improve(t)
return t
```

- Details to Fill in:
  1. How to select initial solution?
  2. How is the neighborhood defined?
  3. How to select a neighbor?
  4. What to do when there are no better neighbors? (exploration vs. exploitation dilemma)
- No single right answer, but many good options

Initial Solutions (TSP as example)

- Assume TSP on fully connected graph
- Nearest Neighbor (NN): At every step, visit closest neighbor that has not already been visited.
  - Euclidean instances: 26% from HK bound - pretty good!
  - Non-euclidean instances: 130-410% from HK bound - not so good
- Greedy Algorithm (GD): TSP version of spanning tree. Keep adding cheapest edge, so long as it doesn't create a node of degree 3, or a cycle.
  - Euglidean: 14%-20% of HKB
  - Non-Euclidean: 100%-280% of HKB

Insertion Algorithms

- All start with small tour, and add nodes such that it is always expanding
- Nearest Addition
  - Given tour, find closest node not on tour(k) to a node on the tour (j)
  - Select a neighbor, i, of j.
  - Delete delete i,j, add j,k and k,i
- Cheapest Insertion

  - Similar to nearest addition, but minimize cost of j,k+k,i instead of just j,k
  - Select node that minimizes c[i,k]+c[k,j]-c[i,j]
  - Slower than nearest addition, but logically would be more accurate
- Farthest Insertion
  - Select node furthest from tour, then use cheapest insertion to find where to add it
  - Euclidean: 16% from HKB

Other 2 Algorithms (category name?)

- MST vs TSP
  - Deleting one edge from HC gives spanning tree!
  - Cheapest HC must be more expensive than MST.
  - MST is lower bound on HC cost
  - Possible to adjust edges of MST to create HC?
  - Traverse tree by always visiting an unvisited neighbor. Backtrack when hit a dead end.
  - When backtracking, don't keep track of nodes that have already been visited.
  - On example graph, go IHGEGHFDADFCFBFI, but don't record the backwards sections
- Eulerian Cycle Approach
  - Eulerian cycle visits each edge once.
  - Eulerian cycle exists if each node of the graph has even degree - trivial to compute
  - Given MST, find all nodes with odd degree
  - Find minimum weighted matching among these odd degree nodes and add edges that are found
  - Now an Eulerian cycle must exist!
  - Find this, and convert it to a HC.

Experimental Methodology

- How to compare algorithm IRL?
- TSPLIB - large, standard collection of hard TSP instances.
- Multiple reasons TSP can be hard
  - For instances from real maps, a->b always shorter than or equal to a->c->b **triangle inequality**

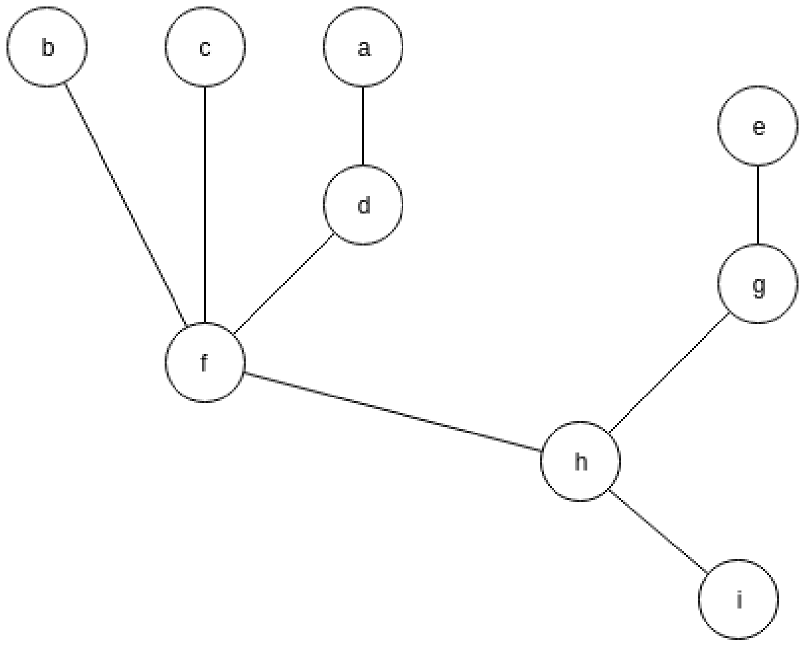


Figure 1: Example Graph

  - Random graphs will not generally satisfy this
  - Triangle inequality helps both heuristics and speed of optimal solutions
- TSPLIB separated into “euclidean"->triangle inequality holds and “non-euclidean"->triangle inequality doesn’t hold
- Using bounds, it is possible to get an upper limit on the difference between a heuristic value and the optimal solution.
- Express TSP in ILP, then use LP bound. Unfortunately, this is long, so approximates used -> Held-Karp bound.

Day 19 Notes

Zach Neveu

June 6, 2019

1 Agenda

- More intro to local search
- Neighborhoods
- Tabu search
- Reading Assigned
- Project due Monday
- Quiz Tuesday

2 Initial Solutions for Local Search

- Eulerian-cycle based algorithm (Christofides Algorithm)
- Gets pretty close to HKB

3 Neighborhood Functions for TSP

- A neighborhood function: select two edges to delete, then add edges to reconnect the cycle
- Maps from one solution to a similar solution
- Gets one neighbor. To get all neighbors, run this function on all pairs of edges.
- $\binom{e}{2}$ total neighbors. Neighborhood called **2-opt** neighborhood
- If current solution better than all 2-opt neighbors, it is a **2-optimal solution**

4 Local Search Algorithms

- Init algorithm
  - Random
  - Nearest Neighbor
  - Greedy

- Steepest descent: Look at 2-opt neighborhood, best neighbor becomes current solution until solution is 2-optimal
- Simple 2-opt helps answer so much!
- 3-Opt: select 3 edges to delete, then add edges to reconnect the cycle. $O(n^3)$ ways to disconnect, 6 ways to put a broken cycle back together. MUCH bigger neighborhood. $O(n^3)$ work for a single step of improvement.
- 2-opt greedy: 4.9%, 3-opt greedy: 3%. Incremental improvement, but MUCH slower to run.
- Local optima problem: These algorithms get permanently stuck in local optima.
- Simplest solution: Repeatedly select random initialization and optimize from each starting point with 2-opt steepest descent.
- Not a particularly a good use of time in practice.
- Revised steepest descent: go to best neighbor, even if it's worse than current solution
- This frequently gets caught in cycles. To fix, just don't go back to already visited solutions
- Tabu list can contain a list of previously visited solutions
- In Tabu search, we select a best neighbor that is not on the tabu list, even if it is worse than current solution. Keep track of champion solution. Stop based on time, time since champion found, improvement in sequential champions small etc.
- Tabu list can actually be quite short, as few as ≈ 10 entries.

Table 1: Initialization Performance

Algorithm	Initial Euclidean	2-Opt	Non-Euclidean Initial	2-Opt
Random	2150	7.9	34500	290
Nearest Neighbor	25	6.6	240	96
Greedy	17.6	4.9	170	70

5 Uniform Graph Partitioning

- Given a graph $G = (V, E)$ with an even number of nodes n, and weighted edges, divide the graph into two groups of nodes with equal size such that the weight of the edges crossing the boundary is minimized.
- Imagine there are two processors. Each node is a job, and the goal is to split the jobs onto the two processors such that each processor has the same amount of work and the amount of data sent between the processors is minimized.
- Neighborhood function: Select a node from each partition and swap them. Equivalent to 2-opt in TSP.
- What to put in the tabu list?

  - Complete solutions. Obvious, and eliminates all cycles with length < 10. Huge amount of into in the queue.
  - Nodes that were recently moved. Prevents going directly backwards. Moves that are made must be left for awhile. Problem with this is it's really restrictive, eliminating possibly valid and good moves. More space efficient than full solutions by far.
  - Store direct pairs of nodes on tabu list. Each node from pair is eligible to be moved again, just not with each other. This is space efficient, and less restrictive. Cycles possible using intermediate variables.

Extensions to Tabu search

- Aspiration-level conditions: ignore the tabu list if a tabu neighbor would be a new champion solution.
- **Intensification**: Focus on solutions that are very similar to previous good solutions. Say a lot of past champions have very similar characteristics. Try freezing these characteristics and focus on changing everything else.
- **Diversification:** Focus on producing unique solutions different from all past solutions.
- Solvers tend to go through phases of diversification then intensification and alternate

Effective Problem Modeling

- Graph Coloring - Minimize # Colors
- Neighborhood function: change the color of one node
- This neighborhood function can have illegal neighbors, and so can also run into the problem of having no legal neighbors.
- Almost all neighbors will have same objective value. We're in Kansas.
- Solution is to change the problem. Instead of minimizing # colors, minimize # conflicts for given number.

Day 20 Notes

Zach Neveu

June 10, 2019

1 Agenda

- Recap of local search
- Variable-depth search
- Simulated Annealing
- Quiz tomorrow
- Project 4 recap

2 Local Search Recap

- Initial solution, made better by looking at neighbors
- TABU search: pick best neighbor even if it is worse
- Stop at time limit, when best answer stops getting better
- Items in TABU list can just be changes that were made, not complete solutions
- TABU list doesn't have to be very long to work
- Aspiration Level conditions: break tabu list rules for really good solutions
- Intensification/Diversification: alternate between searching for best in local area vs. moving to another area.

3 Variable Depth Search

- Consider uniform graph partitioning problem again.
- Break a graph into groups with equal number of nodes
- Minimize weight of inter-group edges
- Size of neighborhood determines how similar neighbors are.
- Larger neighborhood = fewer steps to optimum, slower steps
- Smaller neighborhood = smaller steps to optimum, faster steps
- Tradeoff between neighborhood sizes.

- Consider a potential swap of (a,b)
- Perform the swap, but mark it as “tentative".
- Swap **gain** $g(a,b)$: the decrease in cost function of making swap (a,b)
- Put both a and b on tabu list: neither can be moved
- Choose (a,b) such that $g(a,b)$ is maximized (steepest descent)
- Repeat until no swaps possible (all on tabu list)
- Let cumulative gain $G(k) = \sum_{i=1}^{k} g(a_i, b_i)$ be the gain after k swaps.
- After $\frac{n}{2}$ swaps, partition is back where it started with all items flip-flopped
- $G(k)$ has maximum at k^* which is somewhere between $\{0, \frac{n}{2}\}$
- Take the first k^* steps to get the next neighbor
- This neighborhood function allows the solver to take larger steps without requiring significantly longer steps
- This is basis for Lin-Kernighan TSP

4 Lin-Kernighan

- 2-opt neighborhoods require $O(n^2)$ - too lengthy
- Consider smaller neighborhood
- Consider HCs as paths. Select an edge to delete. Reconnect end of path to first side of deleted edge. Essentially, take a suffix of the path and flip it. Only single way to reconnect, n ways to delete an edge, $O(n)$ neighborhood.
- 2 tabu lists.
  - First list: edges that are added. Once an edge is added, it cannot be deleted.
  - Second list: deleted edges. Once an edge is deleted, it cannot be added.
- Use variable-depth search as optimization technique. Using tabu lists, make up to $\frac{n}{2}$ moves. Look find best out of these solutions and use it as the next solution.
- Turns out this is really effective! Long standing TSP champion algorithm. 1-2% from bound for Euclidean instances.
- Key concepts used: restrictive tabu list, steepest descent, variable depth search, small neighborhood

```
# Local search standard form (before tabu)
def local_search():
    x = initial_soln()
    done = False
    while not done:
        xp = minimum(c(Neighbors(x))
        if c(xp) < c(x):
            x = xp
        else
            done = True

# local search tabu search form
x = initial_soln()
xbest = x
k = 0
while xxx:
    xp = generate(N(x))

    # true if xp better than x
    # also true if xp worse than x up to a point
    if c(xp) - c(x) < t(k)
        x = xp
        if c(x) < c(xp)
            xbest = x
    k += 1
```

- If $t(k) = 0$, then this gets stuck in any local optimum
- If $t(k) = \infty$, then first neighbor always visited
- $t(k)$ controls tradeoff between finding optimum and exploring.
- Start with large value of $t(k)$, anneal to reach small $t(k)$ by the end.
- $t(k) \geq t(k+1)$, $lim_{k\to\infty} t(k) = 0$. Threshold Accepting.

5 Simulated Annealing

- Instead of $t(k)$ having fixed value, we let $t(k)$ be a random variable > 0
- Consider

$$p(N_i) = \begin{cases} 1 & c(xp) \leq c(x) \\ e^{\frac{c(x)-c(xp)}{d_k}} & c(xp) > c(x) \end{cases}$$

- Better solutions always excepted
- Worse potentially excepted. Worse solutions have smaller probability.
- d_k adjusts curve of falloff. Larger d_k means higher chance of going to worse neighbors. Smaller d_k means smaller chance of visiting worse neighbors.
- d_k varies with k. Can start large to diversify, then get smaller to intensify
- **cooling schedule:** how fast does d_k get smaller?

- Name: from quantum physics, simulating particle motion. d_k is temperature. Cooling schedule has very physical meaning here!

Simulated Annealing TSP

- SA1 algorithm
- Neighborhood function: 2-opt
- Set $d_0 \approx \infty$ - accept like 95% of answers
- set $d_k = d_{k-1}^{0.95}$
- Temperature length: $N(N-1)$ - how long to spend at each temperature
- Results (see 1): SA1 slow! Can be improved greatly using 2-opt afterwords.
- How to speed up?
- Neighborhood pruning. Smaller neighborhood → faster annealing → faster result.
  - Neighborhood: select a random edge to delete, select one of its 20 closest neighbors to delete. Then reconnect. This takes us from $0(n^2) \rightarrow O(n)$
  - Temp length = $\alpha * 20n$
- Low-temperature start, and don't use random initial solution
  - Start with good heuristic solution
  - Lower initial temperature
  - Speeds up search

Table 1: SA1 Results: Size vs. Algorithm

N=	100	316	1000
SA1	5.2(12.4s)	4.1(188s)	4.1(3170s)
SA1+2-opt	3.4	3.7	4.0
2-opt	4.5(0.03s)	4.8 (0.09s)	4.9 (0.34s)
3-opt	2.5(0.04s)	2.5 (0.1s)	3.1 (0.4s)
LK	1.5(0.06)	1.7(0.2)	2.0(0.77s)