Announcements

- Final Exam March 19th

Last Time

- How do you deal with NP-Hard problems?

Local Search

Many optimization problems have a structure where solutions “nearby” a good solution will likely also be good.

This leads to a natural algorithmic idea:

- Find an OK solution
- Search nearby for better solutions
- Repeat

Local Search

```
LocalSearch(f)
\\ Try to maximize f(x)
   x ← Random initial point
   Try all y close to x
      If f(y) > f(x) for some y
         x ← y
         Repeat
      Else Return x
```

Today

- Local Search
- Approximation Algorithms

MAXCUT

Problem: Given a graph G find a way to color the vertices of G black and white so that as many edges as possible have endpoints of different colors.

This is NP-Hard.

Question: MAXCUT

What is the size of the MAXCUT of the graph below?

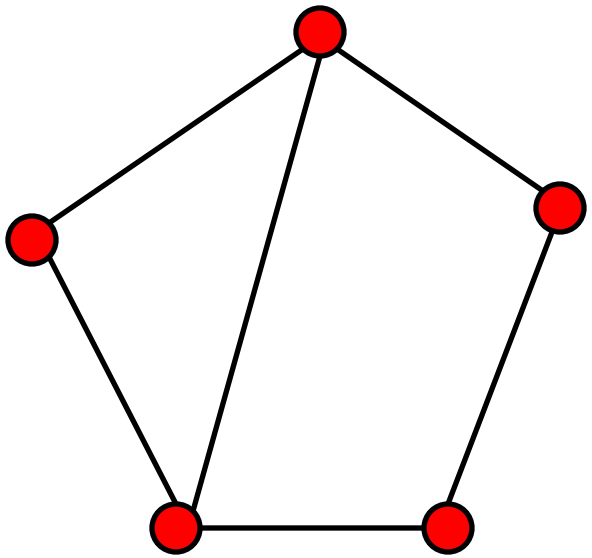
A) 2

B) 3

C) 4

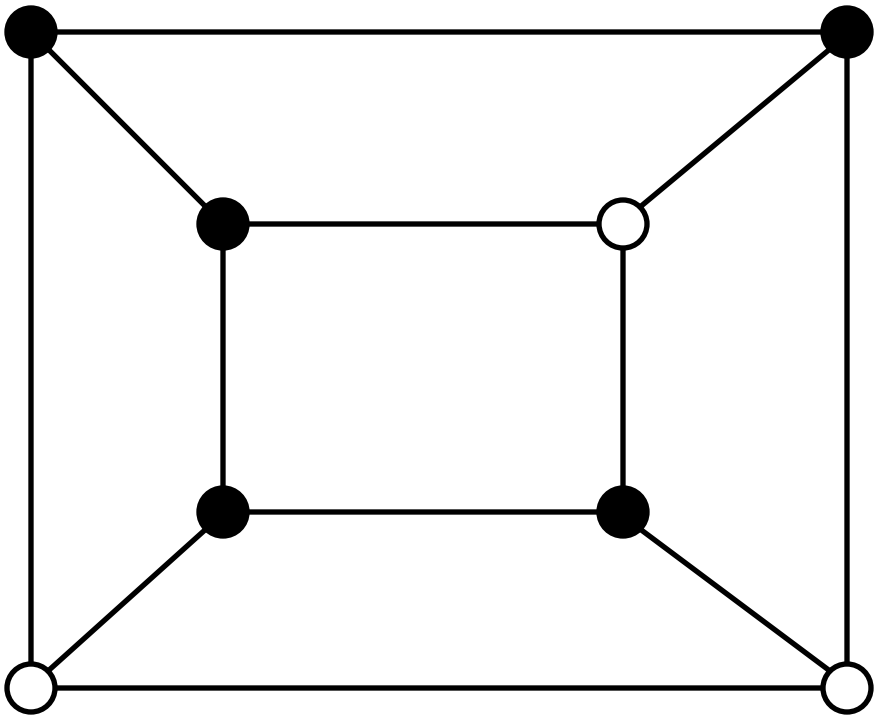
D) 5

E) 6

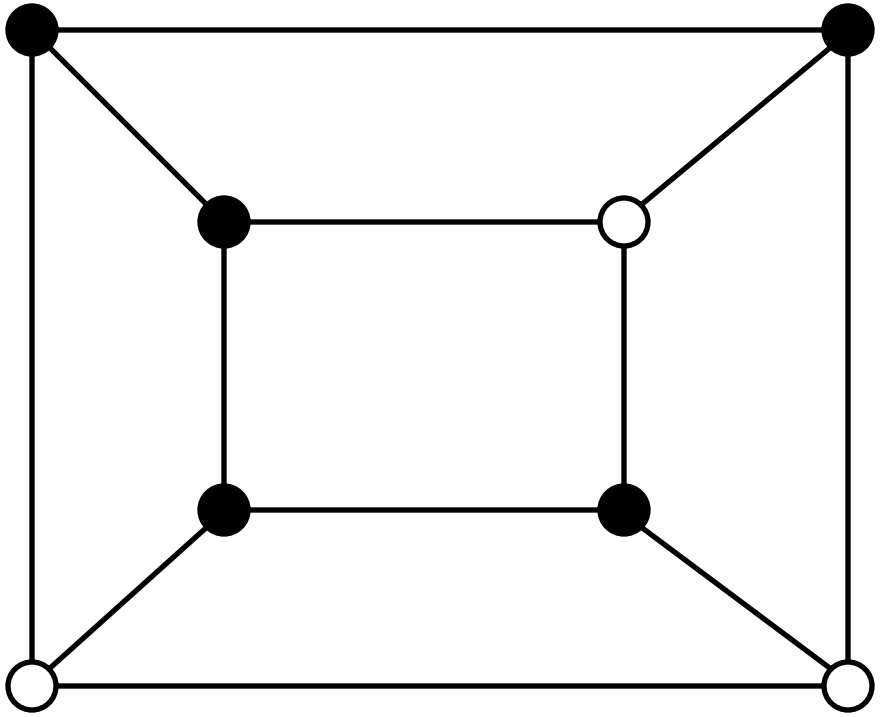


MAXCUT: Local Search

If possible recolor one vertex at a time for maximum improvement.

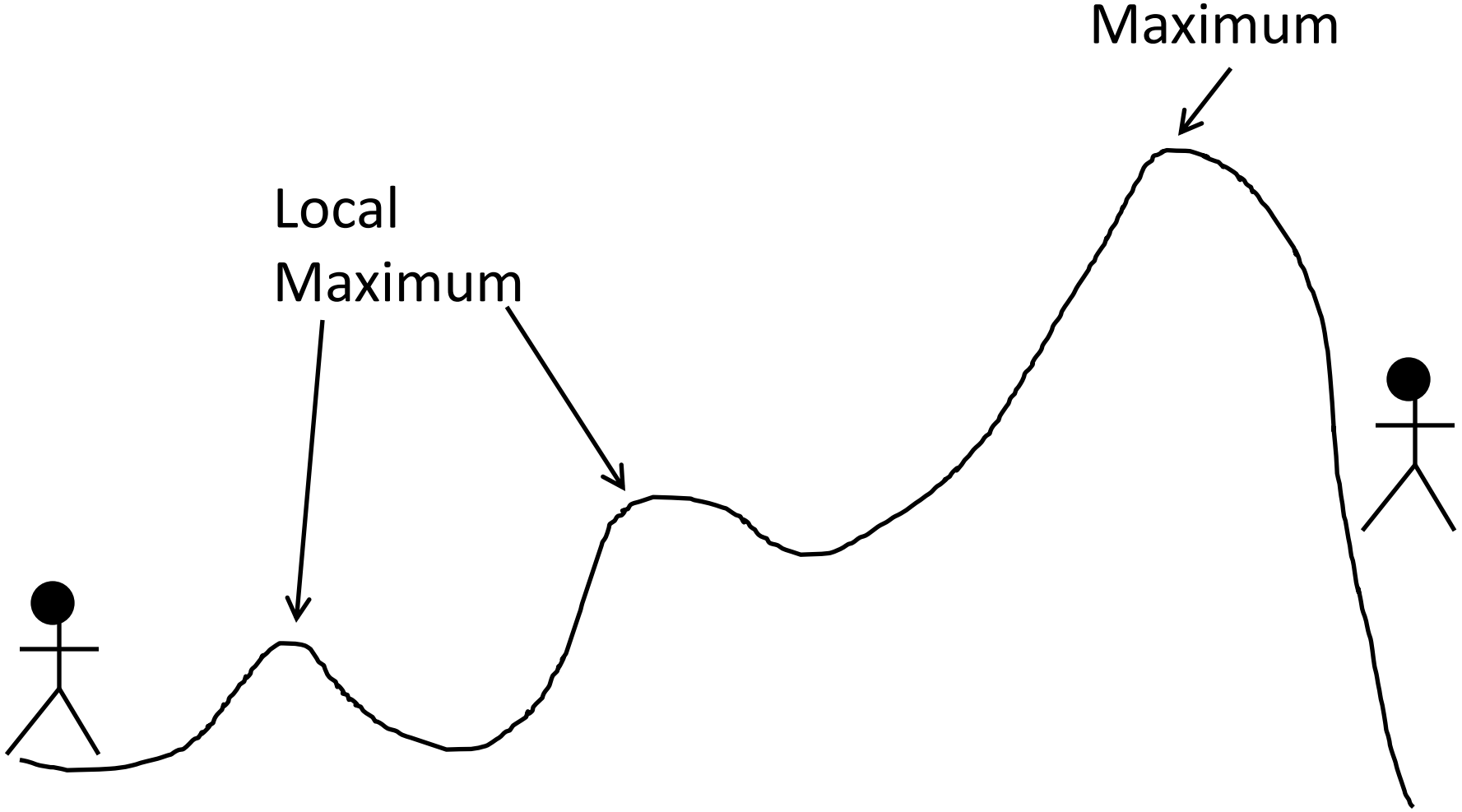


Problem



Stuck!

Local Maxima



How to Get Unstuck

- Randomized Restart
  - If you try many starting points, hopefully, you will find one that finds you the true maximum.
- Expand Search Area
  - Look for changes to 2 or 3 vertices rather than 1.
    - Larger area means harder to get stuck
    - Larger area also takes more work per step
- Still no guarantee of finding the actual maximum in polynomial time.

Simulated Annealing

- At the start of algorithm take big random steps.
  - Hopefully, this will get you onto the right “hill”.
- As the algorithm progresses, the “temperature” decreases and the algorithm starts to fine tune more precisely.
- Works well in practice on a number of problems.

MAXCUT Minimal Value

Look back at local search for MAXCUT.

- Swap a vertex if most of its neighbors are the same color.
- At the end of the algorithm most of a vertices neighbors are the opposite color.
- At the end of the algorithm at least half of the edges are cut.
- Get cut of size at least $|E|/2$, but optimum at most $|E|$.

Approximation Algorithms

An α-approximation algorithm to an optimization problem is a (generally polynomial time) algorithm that is guaranteed to produce a solution within an α-factor of the best solution.

Our local search algorithm for MAXCUT is a 2-approximation algorithm.

Often approximation algorithms can produce good enough solutions.

Vertex Cover

Often greedy algorithms can give approximation algorithms.

Problem (Vertex Cover): Given a graph G find a set S of vertices so that every edge of G contains a vertex of S and so that |S| is as small as possible.

Also, NP-Hard.

Question Vertex Cover Example

What is the size of the smallest vertex cover in the graph below?

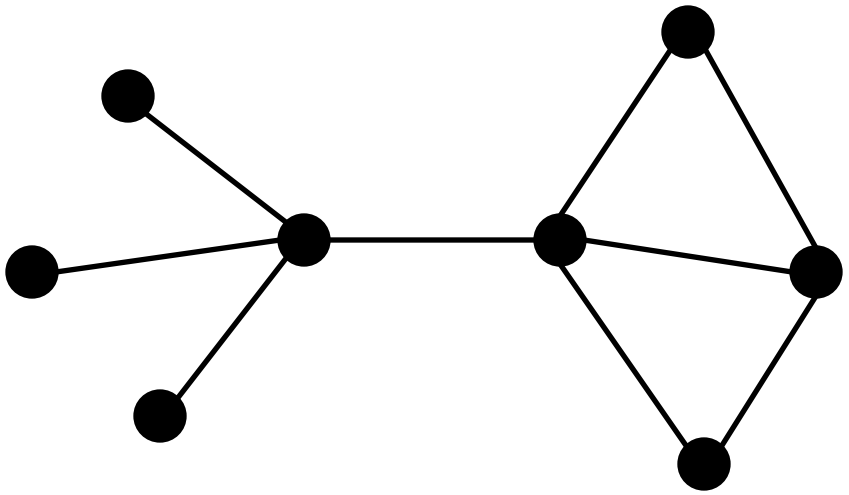
A) 1

B) 2

C) 3

D) 4

E) 5

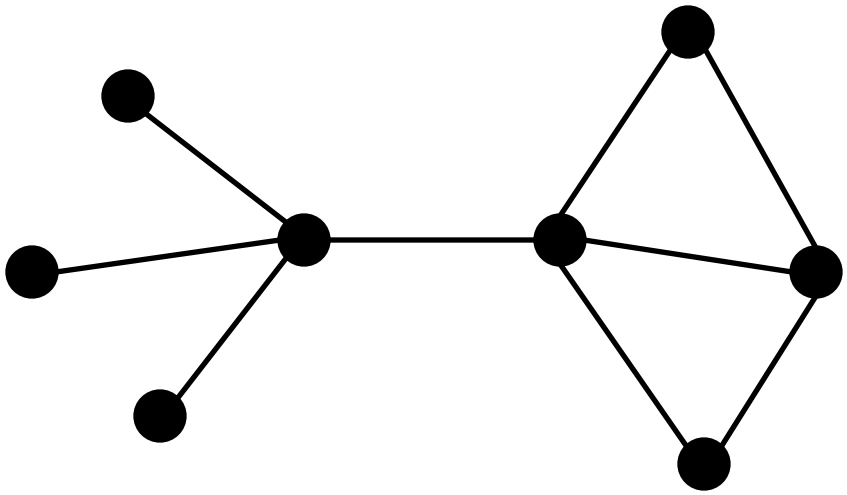


Greedy Algorithm

```
GreedyVertexCover(G)
   S ← {}
   While(S doesn't cover G)
      (u,v) ← some uncovered edge
      Add u and v to S
   Return S
```

Simple and fast.

Example



Analysis

Algorithm finds k edges and 2k vertices.

- Edges are vertex-disjoint.
- Any cover must have at least one vertex on each of these edges.
- Optimum cover has size at least k.
- We have a 2-approximation.

Knapsack

Even though general knapsack is NP-Hard, we have a polynomial time algorithm if all weights are small integers (or more generally small integer multiples of some common value).

Since everything can be rounded to small integers, we have an algorithm idea.

Small Values

Actually rounding the weights doesn't quite work. It gives you sets which almost fit in the sack.

Instead, we want to round the values of the items and for this, we need a new algorithm.

Dynamic Program

Let $\text{Lightest}_{\leq k}(V)$ be the weight of the lightest collection of the first k items with total value V.

We have a recursion

$\text{Lightest}_{\leq k}(V) = \min\{\text{Lightest}_{\leq k-1}(V), \text{Wt}(k) + \text{Lightest}_{\leq k-1}(V - \text{Val}(k))\}$

This gives a DP.

#subprobs = [Total Value][#items]

Time/Subproblem = O(1).

Approximation Algorithm

1) Throw away items that don't fit in sac.
2) Let V_0 be highest value of item.
3) Round each item's value to closest multiple of δV_0.
4) Run the small integer values DP.

Runtime: Values integer multiples of δV_0. Total value at most $[\#items]V_0 = ([\#items]/\delta)\ \delta V_0$.

Total runtime $O([\#items]^2/\delta)$.

Approximation Analysis

Optimal value at least V_0.

Rounding changes the value of any set of items by at most [#items]δV_0.

The solution we find is at least as good as the optimal after round.

Our solution is within [#items]δV_0 of optimal.

Combining

Let $\delta = \varepsilon/[\#\text{items}]$.

$OPT \geq V_0$.

Our solution is at most εV_0 worse.

Have a $(1+\varepsilon)$-approximation algorithm.

Runtime = $O([\#\text{items}]^3/\varepsilon)$

For any $\varepsilon > 0$, have a $(1+\varepsilon)$-approximation in polynomial time. (known as a PTAS).