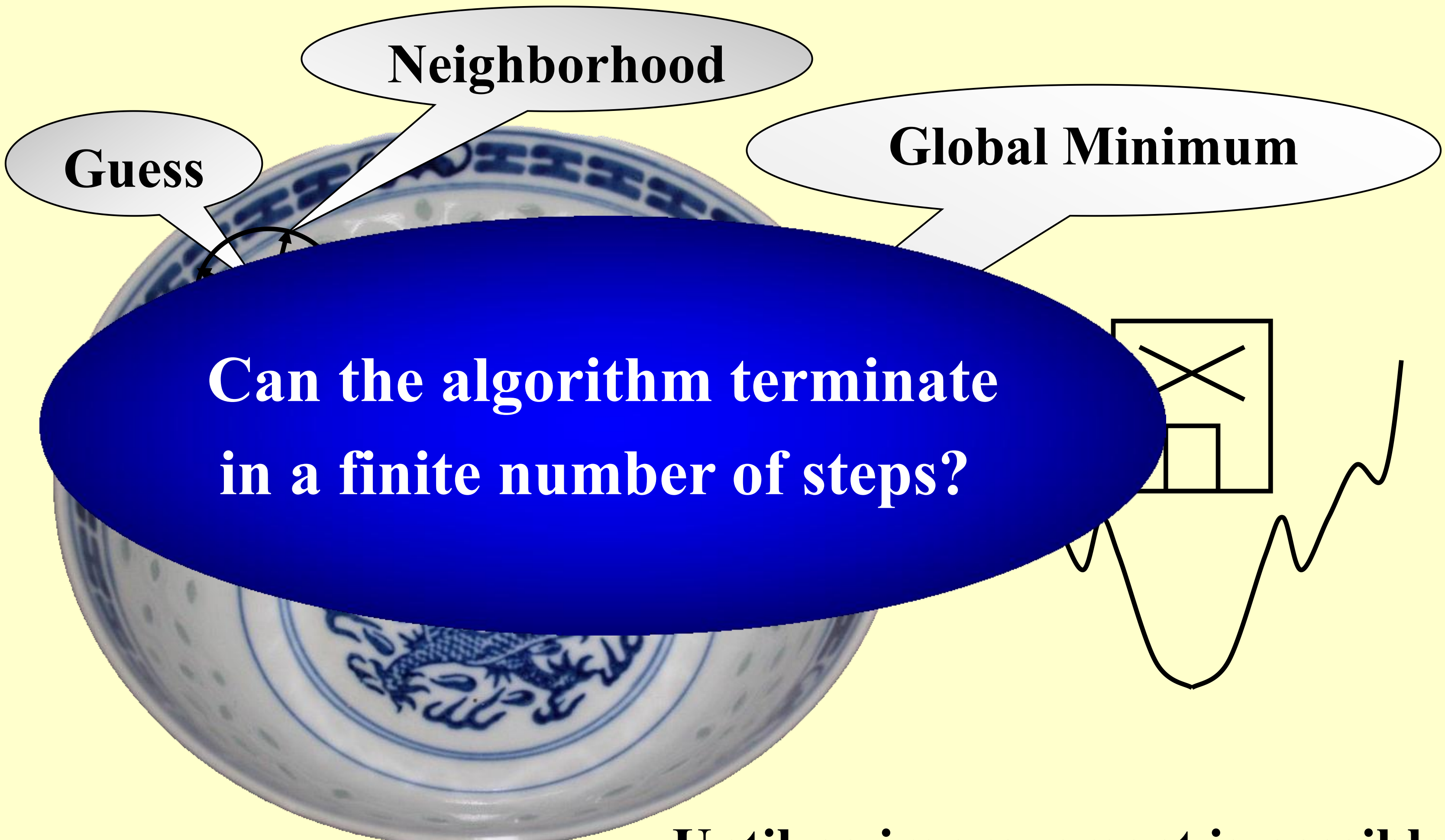
Local Search

Solve problems *approximately*

—— aims at a local optimum



Until no improvement is possible

Framework of Local Search

☞ **Local**

- **Define *neighborhoods* in the feasible set**
- **A local optimum is a best solution in a neighborhood**

☞ **Search**

- **Start with a feasible solution and search a better one within the neighborhood**
- **A local optimum is achieved if no improvement is possible**

Neighbor Relation

☞ S ~ S' : S' is a *neighboring solution* of S – S' can be obtained by a small modification of S.

☞ N(S): *neighborhood* of S – the set { S': S ~ S' }.

```
SolutionType Gradient_descent()
{   Start from a feasible solution S ∈ FS ;
    MinCost = cost(S);
    while (1) {
        S’ = Search( N(S) ); /* find the best S’ in N(S) */
        CurrentCost = cost(S’);
        if ( CurrentCost < MinCost ) {
            MinCost = CurrentCost;    S = S’;
        }
        else  break;
    }
    return S;
}
```

〖Example〗 The Vertex Cover Problem.

- ❖ Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?

- ❖ Vertex cover problem: Given an undirected graph G = (V, E). Find a ***minimum*** subset S of V such that for each edge (*u*, *v*) in E, either *u* or *v* is in S.

☞ Feasible solution set $\mathcal{FS}$: all the vertex covers.

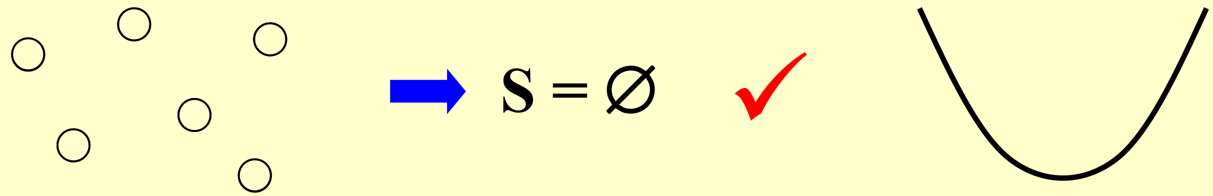
☞ cost(S) = | S |

☞ S ~ S':

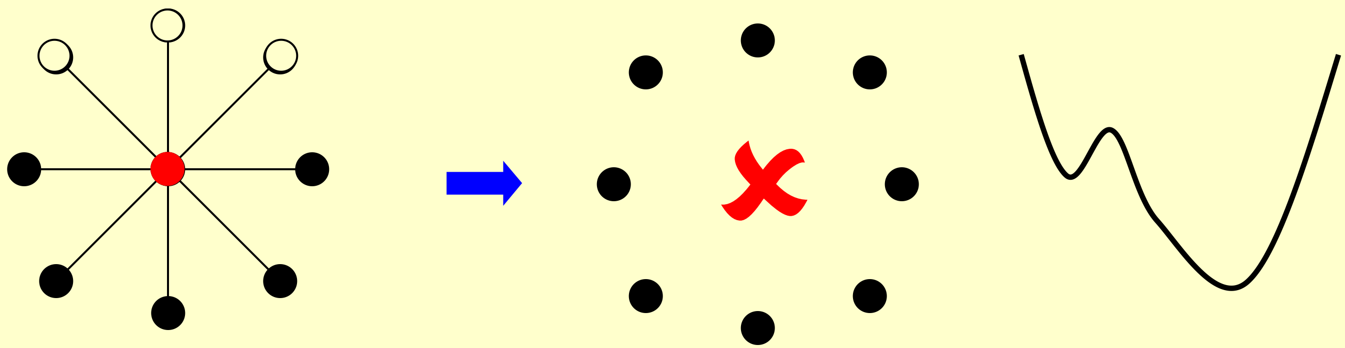
Each vertex cover S has at most |V| neighbors.

☞ Search: Start from S = V; delete a node and check if S' is a vertex cover with a smaller cost.

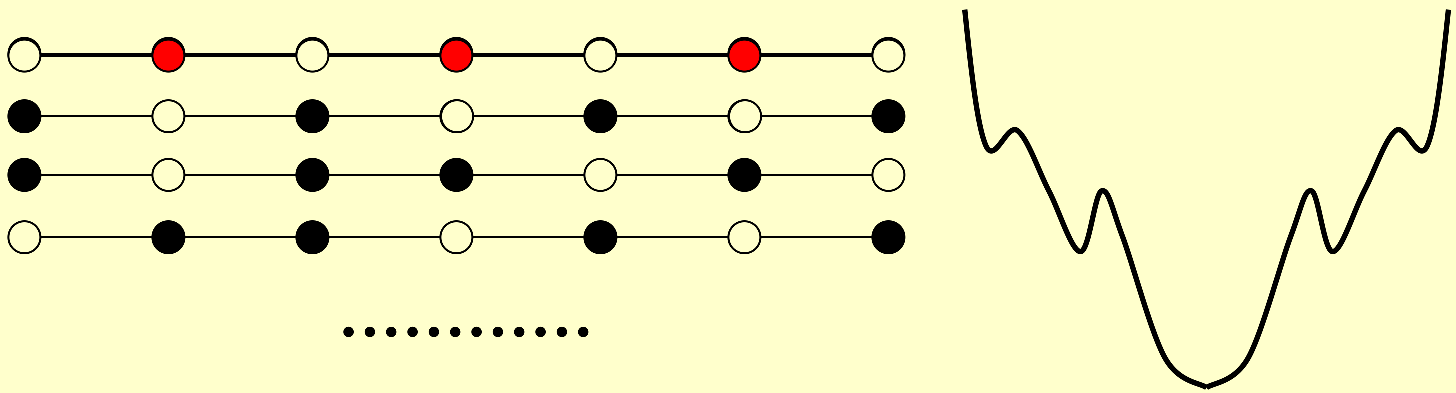
Case 0:



Case 1:



Case 2:



Discussion 17:

Can you give another case in which gradient descent doesn't work?

Try to improve …

☞ The Metropolis Algorithm

```
SolutionType Metropolis()
{   Define constants k and T;
    Start from a feasible solution S ∈ FS ;
    MinCost = cost(S);
    while (1) {
       S' = Randomly chosen from N(S);
       CurrentCost = cost(S');
       if ( CurrentCost < MinCost ) {
          MinCost = CurrentCost;    S = S';
       }
       else {
          With a probability e^{-Δcost/(kT)} , let S = S';
          else  break;
       }
    }
    return S;
}
```

Adding is allowed

☞ Simulated Annealing

The material is cooled *very gradually* from a high temperature, allowing it enough time to reach equilibrium at a succession of intermediate lower temperatures.

Cooling schedule: $T = \{ T_1 , T_2 , \ldots \}$

〖Example〗 Hopfield Neural Networks

Graph G = (V, E) with integer edge weights w (positive or negative).

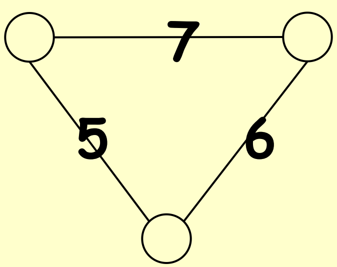
If $w_e < 0$, where $e = (u, v)$, then u and v want to have the *same state*;
if $w_e > 0$ then u and v want *different states*.

The absolute value $|w_e|$ indicates the *strength* of this requirement.

± 1

Output: A *configuration S* of the network –
an assignment of the state s_u to each node u

There may be no configuration that respects the requirements imposed by all the edges.



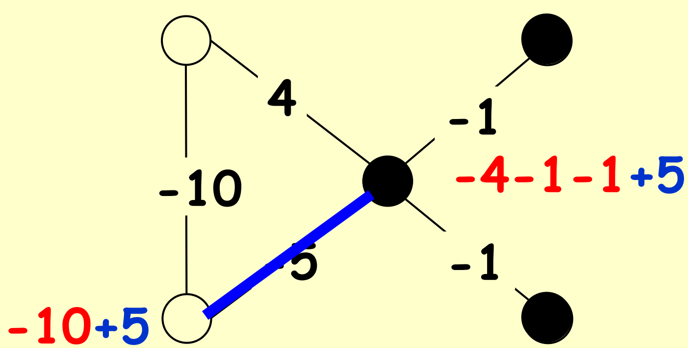
☞ Find a configuration that is *sufficiently good*.

【**Definition**】 **In a configuration S, edge $e = (u, v)$ is *good* if $w_e s_u s_v < 0$ ($w_e < 0$ iff $s_u = s_v$); otherwise, it is *bad*.**

【**Definition**】 **In a configuration S, a node u is *satisfied* if the weight of incident good edges $\geq$ weight of incident bad edges.**

$$\sum_{v:e=(u,v)\in E} w_e s_u s_v \leq 0$$

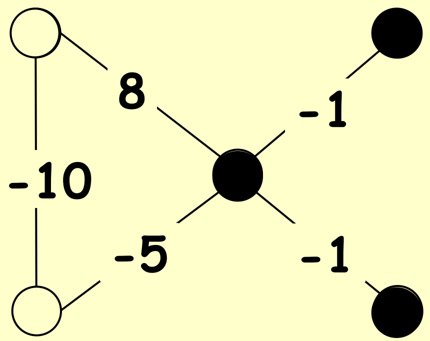
【**Definition**】 **A configuration is *stable* if all nodes are satisfied.**



✓ ***Does a Hopfield network always have a stable configuration, and if so, how can we find one?***

☞ State-flipping Algorithm

```
ConfigType State_flipping()
{
    Start from an arbitrary configuration S;
    while ( ! IsStable(S) ) {
        u = GetUnsatisfied(S);
        s_u = - s_u;
    }
    return S;
}
```



Will it *always* terminate?

Claim: **The state-flipping algorithm terminates at a stable configuration after *at most* $W = \Sigma_e|w_e|$ iterations.**

Proof: **Consider the measure of progress**

$$\Phi(S) = \Sigma_{e \text{ is } good} |w_e|$$

When u flips state (S becomes S'):

- **all good edges incident to u become bad**
- **all bad edges incident to u become good**
- **all other edges remain the same**

$$\Phi(S') = \Phi(S) - \sum_{\substack{e:e=(u,v)\in E \\ e \text{ is bad}}} |w_e| + \sum_{\substack{e:e=(u,v)\in E \\ e \text{ is good}}} |w_e|$$

Clearly $0 \le \Phi(S) \le W$ ∎

Related to Local Search

- ☞ Problem: To *maximize* Φ.
- ☞ Feasible solution set $\mathcal{FS}$: configurations
- ☞ S ~ S': S' can be obtained from S by flipping a single state

Claim: Any local maximum in the state-flipping algorithm to maximize Φ is a stable configuration.

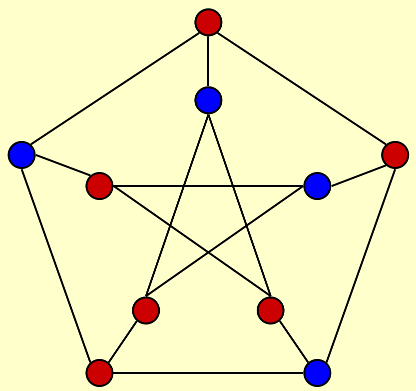
Is it a polynomial time algorithm?

Still an open question: to find an algorithm that constructs stable states in time polynomial in *n* and logW (rather than *n* and W), or in a number of primitive arithmetic operations that is polynomial in *n* alone, independent of the value of W.

〖Example〗 The Maximum Cut Problem.

❖ **Maximum Cut problem:** Given an undirected graph G = (V, E) with positive integer edge weights w_e, find a node partition (A, B) such that the total weight of edges crossing the cut is maximized.

$$w(A,B) := \sum_{u \in A,\, v \in B} w_{uv}$$



- **Toy application**
  - n activities, m people.
  - Each person wants to participate in two of the activities.
  - Schedule each activity in the morning or afternoon to maximize number of people that can enjoy both activities.
- **Real applications** Circuit layout, statistical physics

Related to Local Search

Single-flip neighborhood

- ☞ Problem: To *maximize* $\Phi(S) = \sum_{e \text{ is good}} |w_e|$
- ☞ Feasible solution set *FS* : any partition (A, B)
- ☞ S ~ S': S' can be obtained from S by moving one node from A to B, or one from B to A.

☝ A special case of Hopfield Neural Network – with w_e all being positive!

```
ConfigType State_flipping()
{
   Start from an arbitrary configuration S;
   while ( ! IsStable(S) ) {
      u = GetUnsatisfied(S);
      s_u = - s_u;
   }
   return S;
}
```

May NOT in polynomial time

- How good is this local optimum?
- Try a better *local*?

- How good is this local optimum?

Claim: Let (A, B) be a local optimal partition and let (A^*, B^*) be a global optimal partition. Then $w(A, B) \geq ½\, w(A^*, B^*)$.

Proof: Since (A, B) is a local optimal partition, for any $u \in A$

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

Summing up for *all* $u \in A$

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)$$

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B)$$

$$w(A^*, B^*) \leq \sum_{\{u,v\} \subseteq A} w_{uv} + \sum_{\{u,v\} \subseteq B} w_{uv} + w(A, B) \leq 2w(A, B)$$

$\blacksquare$

- [Sahni-Gonzales 1976] There exists a 2-approximation algorithm for MAX-CUT.

$$\min_{0\le\theta\le\pi} \frac{\pi}{2}\frac{1-\cos\theta}{\theta}$$

- [Goemans-Williamson 1995] There exists a 1.1382-approximation algorithm for MAX-CUT.

- [Håstad 1997] Unless P = NP, no 17/16 approximation algorithm for MAX-CUT.

1.0625

- May NOT in polynomial time

☝ stop the algorithm when there are no "*big enough*" improvements.

Big-improvement-flip: Only choose a node which, when flipped, increases the cut value by at least

$$\frac{2\varepsilon}{|V|} w(A, B)$$

Claim: Upon termination, the big-improvement-flip algorithm returns a cut (A, B) so that

$$(2 + \varepsilon)\, w(A, B) \geq w(A^*, B^*)$$

Claim: The big-improvement-flip algorithm terminates after at most $O(n/\varepsilon \log W)$ flips.

- **Try a better *local* ?**

☞ **The neighborhood of a solution should be rich enough that we do not tend to get stuck in bad local optima; but**
☞ **the neighborhood of a solution should not be too large, since we want to be able to efficiently search the set of neighbors for possible local moves.**

Single-flip ➡ *k*-flip ➡ $\Theta(n^k)$ for searching in neighbors

[Kernighan-Lin 1970] *K-L heuristic*

Step 1: make 1-flip as good as we can – $O(n)$ ➡ (A_1, B_1) and v_1

Step k: make 1-flip of an *unmarked* node as good as we can – $O(n-k+1)$ ➡ (A_k, B_k) and $v_1 \dots v_k$

Step n: $(A_n, B_n) = (B, A)$

Neighborhood of $(A, B) = \{ (A_1, B_1), \dots, (A_{n-1}, B_{n-1}) \}$ $\quad O(n^2)$

Reference:

Algorithm Design：Ch.12，p.661-706；*Jon Kleinberg, Eva Tardos, Addison Wesley, 2005*