Simulated Annealing for the Maximum Clique Problem

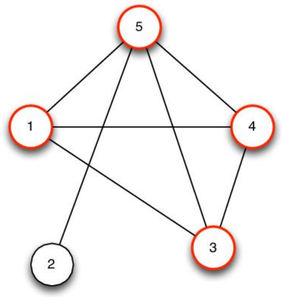
Brendan Banfield, Aaron Banse, Kellen Knop, Sophie Quinn

Carleton

The Maximum Clique Problem

A graph $G=(V, E)$ is a collection of vertices V and edges E connecting vertices. We can use a graph's adjacency matrix A_G to see if an edge exists; $A_G(v_1, v_2) = 1$ if there is an edge connecting v_1 to v_2 and is 0 otherwise.

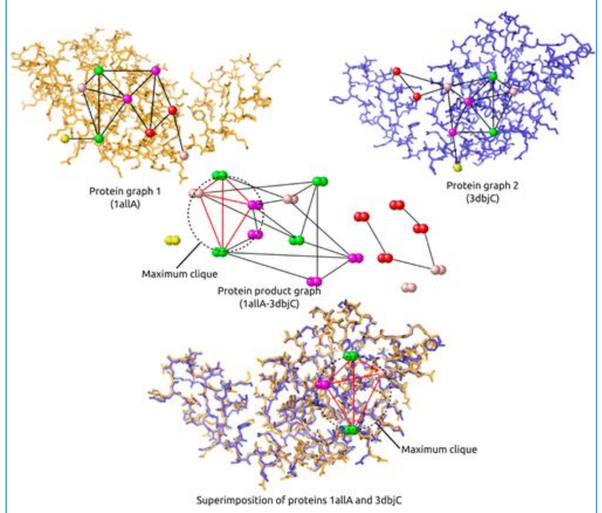
A *clique* is a subset of nodes in an undirected, unweighted graph that are all connected to each other. The maximum clique in a graph is the largest clique in the graph.



The red vertices form a clique since each pair of vertices in the clique is connected with an edge. Since there is no larger clique in the graph, the red vertices form the maximum clique.

Why We Care

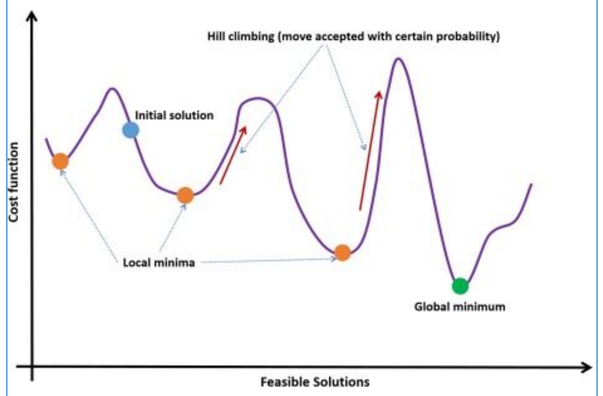
- Finding the maximum clique is NP-hard, so if we could solve it quickly, we would be able to solve many other problems efficiently
- While that probably isn't possible, studying the problem can still lead to valuable insights
- There are specific use cases for the maximum clique including protein structure analysis and image segmentation



Above: An example application of the maximum clique problem. Protein structures are converted to graphs. The graphs from two different proteins are combined, and the largest clique in the new graph is the largest structure the two proteins share. This can be used to compare and categorize proteins. Credit Depolli et al, [2].

Simulated Annealing

- Simulated annealing is an algorithm strategy that can be applied to many different problems.
- It's inspired by metallurgical annealing, where a metal is heated and then cools incrementally, decreasing in energy
- In simulated annealing, some *objective function* is used instead of energy
- You begin in a randomized state, and then make incremental random changes that lower the objective function over time.
- The algorithm needs to occasionally make "bad" decisions that increase the cost function. This is required to escape local minima and find the overall best solution, and is called *hill climbing*
- Initially, simulated annealing algorithms will make many choices that increase the objective function, but over time they will "cool" they become more greedy
- Since simulated annealing is random, it won't reach the correct answer every time. It makes up for this by having a much lower runtime than exact algorithms



Above: Simulated annealing for a generic problem. Choosing poor short term choices that increase the cost function is required to "hill climb" out of local minima to find the global optimal solution.

Simulated Annealing for Maximum Clique

I implemented a simulated annealing algorithm for the maximum clique problem created by Geng et al. in [2]. The algorithm takes a graph G with adjacency matrix A_G, and a clique size k as a parameter and searches purely for cliques of that size; it will never find larger or smaller cliques. To do this, it maintains a list C comprising the k vertices in the graph that are "in the clique", even if the vertices don't actually form a clique. The objective function is the total number of missing edges between vertices in the clique:

$$F(G, C) = \sum_{v_1 \in C} \sum_{\substack{v_2 \in C \\ v_2 \neq v_1}} (1 - A_G(v_1, v_2))$$

where $A_G(v_1, v_2) = 1$ if v_1 and v_2 share an edge and is 0 otherwise.

The "clique" C is initially likely to be a poor choice. The algorithm repeatedly considers swapping a vertex in C with one not in C. It performs the swap if

Algorithm Test Results

The simulated annealing algorithm along with two exact algorithms, Bron-Kerbosch and branch and bound, were tested on a subset of the DIMACS graphs, benchmark graphs with known maximum cliques. Algorithms were stopped if they did not finish after 2 hours, and simulated annealing had 10 trials on each graph.

Graph	B-K runtime (s)	B&B runtime (s)	Sim. ann. runtime (s)	Sim. ann. success rate
c-fat200-1	0.0019	0.004	1.1	100%
c-fat500-1	0.0085	0.0096	7.6	100%
johnson16-2-4	16	4.1	0.0065	100%
johnson32-2-4	Timed out	Timed out	0.25	100%
keller4	15	5.5	2.2	100%
keller5	Timed out	Timed out	180	10%
keller6	Timed out	Timed out	1300	0%
hamming10-2	Timed out	320	770	90%
hamming8-2	Timed out	1.7	30	100%
san200_0.7_1	Timed out	7.0	9.3	0%
san400_0.5_1	Timed out	7.6	1.2	0%
san400_0.9_1	Timed out	Timed out	94	0%
sanr200_0.7	340	160	12	90%
sanr400_0.5	140	200	92	0%
san1000_0.5	Timed out	1300	1.5	0%
brock200_1	2100	740	45	20%
brock400_1	Timed out	Timed out	130	0%
brock800_1	Timed out	Timed out	250	0%
p_hat300-1	0.21	1.2	5.5	100%
p_hat500-1	2.3	13	9.7	100%
p_hat700-1	12	53	86	50%
p_hat1000-1	60	290	41	90%
p_hat1500-1	690	3600	370	0%
MANN_a27	Timed out	Timed out	95	0%
MANN_a45	Timed out	Timed out	560	0%

For small graphs. simulated annealing runs slower than exact solutions. On many graphs, simulated annealing keeps up with or outperforms exact solutions. For a few graphs (e.g. johnson32-2-4) simulated annealing finds optimal solutions quickly when exact algorithms take hours.

The algorithm can be tweaked to run faster at the cost of accuracy on certain graphs.

Sources:
[1] Miroslaw Blocho, Chapter 4 - Heuristics, metaheuristics, and hyperheuristics for rich vehicle routing problems, In Intelligent Data-Centric Systems, Smart Delivery Systems, Elsevier, 2020, Pages 101-156, ISBN 9780128157152, https://doi.org/10.1016/B978-0-12-815715-2.00009-9.

[2] Matjaž Depolli, Janez Konc, Kati Rozman, Roman Trobec, and Dušanka Janežič. Journal of Chemical Information and Modeling 2013 53 (9), 2217-2228. DOI: 10.1021/ci4002525

The Algorithm

```
Function ObjectiveFunction(A_G, C):
  numMissingEdges ← 0;
  for each unique pair of vertices (v_1, v_2) in C:
    # if there isn't an edge between them, increment numMissingEdges
    if A_G(v_1, v_2) is 0 then:
      numMissingEdges ← numMissingEdges + 1;
    end if
  end for
return numMissingEdges

Function SimulatedAnnealing(G, A_G, k, τ_0, τ_f, α):
  τ ← τ_0;
  n ← the number of vertices in G;
  C ← the k highest-order vertices of G;
  # higher order vertices are more likely to be in the maximum clique,
  # so this is generally a better than average starting set

  while τ > τ_f do:
    currentScore ← ObjectiveFunction(A_G, C);
    v_1 ← a random vertex in C;
    v_2 ← a random vertex in G but not in C;
    remove v_1 from C;
    add v_2 to C;
    newScore ← ObjectiveFunction(A_G, C);
    ΔF ← newScore - currentScore;
    # ΔF is the change in number of missing edges in C
    i ← 0
    while (i < 8*n) and (ΔF > 0) do:
      # if the edge makes our graph worse, pick again
      # (up to 8n times)
      remove v_2 from C;
      v_2 ← a random vertex in G but not in C;
      add v_2 to C;
      newScore ← ObjectiveFunction(A_G, C);
      ΔF ← newScore - currentScore;
      i ← i + 1;
    end while
    if newScore is 0 then:
      # no missing edges–found a clique of size k!
      return C;
    end if


    if ΔF > 0 then:
      # if the swap makes our "clique" C worse, we
      # randomly choose whether to revert it
      r ← Random(0, 1);
      p ← e^(-ΔF * τ);
      if r > p then:
        remove v_2 from C;
        add v_1 to C;
      end if
    end if

    # decrease the likelihood of making choices that
    # lower the score of C in the future (since α < 1)
    τ ← τ * α
  end while

# failed to find a clique of size k
return C;
```