Dynamic programming

- Solves a complex problem by breaking it down into subproblems
- Each subproblem is broken down recursively until a trivial problem is reached
- Computation itself is not recursive: problems are solved from simple to more complex
  - Trivial problems are solved first
  - More complex solutions are composed from the simpler solutions already computed

Dynamic programming

- Applicable efficiently when
  - Composing more complex solutions from subproblems solutions is fast (linear time)
  - Subproblems are *overlapping*: a single solution is required to solve several other subproblems
    - Has a clear advantage over recursion
  - Has *optimal substructure*
    - Each level of subproblems is only slightly more complex than the lower level
    - See *Principle of optimality*, *Bellman equation* etc.

Polynomial time algorithms

- Floyd-Warshall algorithm
- CYK algorithm
- Levenshtein distance
- Viterbi algorithm
- Several string algorithms

Exponential time algorithms

- Useful for many problems where search space is superexponential in the input size n
  - Permutation problems, $O^*(n!)$
    - Example: Travelling salesman problem
  - Partition problems, $O^*(n^n)$
    - Example: Graph coloring problem
- Typically solved dynamically by identifying subproblems on subsets of the original problem
  - The number of subsets is ”only” exponential in n

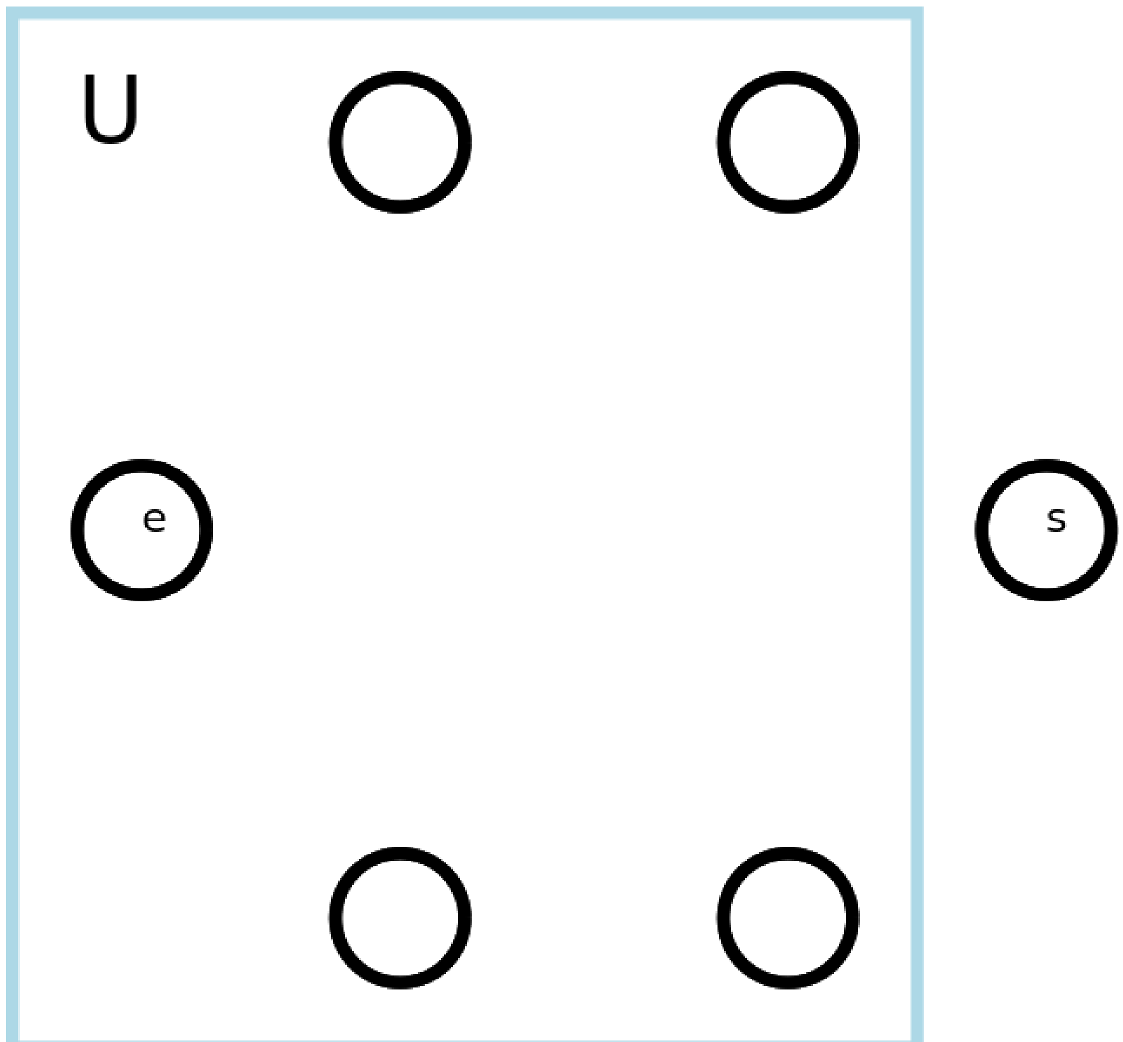
Travelling salesman problem

- Given an undirected weighed graph (V, E) of n vertices, find a cycle of minimum weight that visits each vertex in V exactly once
- A permutation problem: brute-force search enumerates all permutations of vertices, running in time $O^*(n!)$
- Associated decision problem is NP-complete
- With dynamic programming we can solve the problem in time $O^*(2^n)$

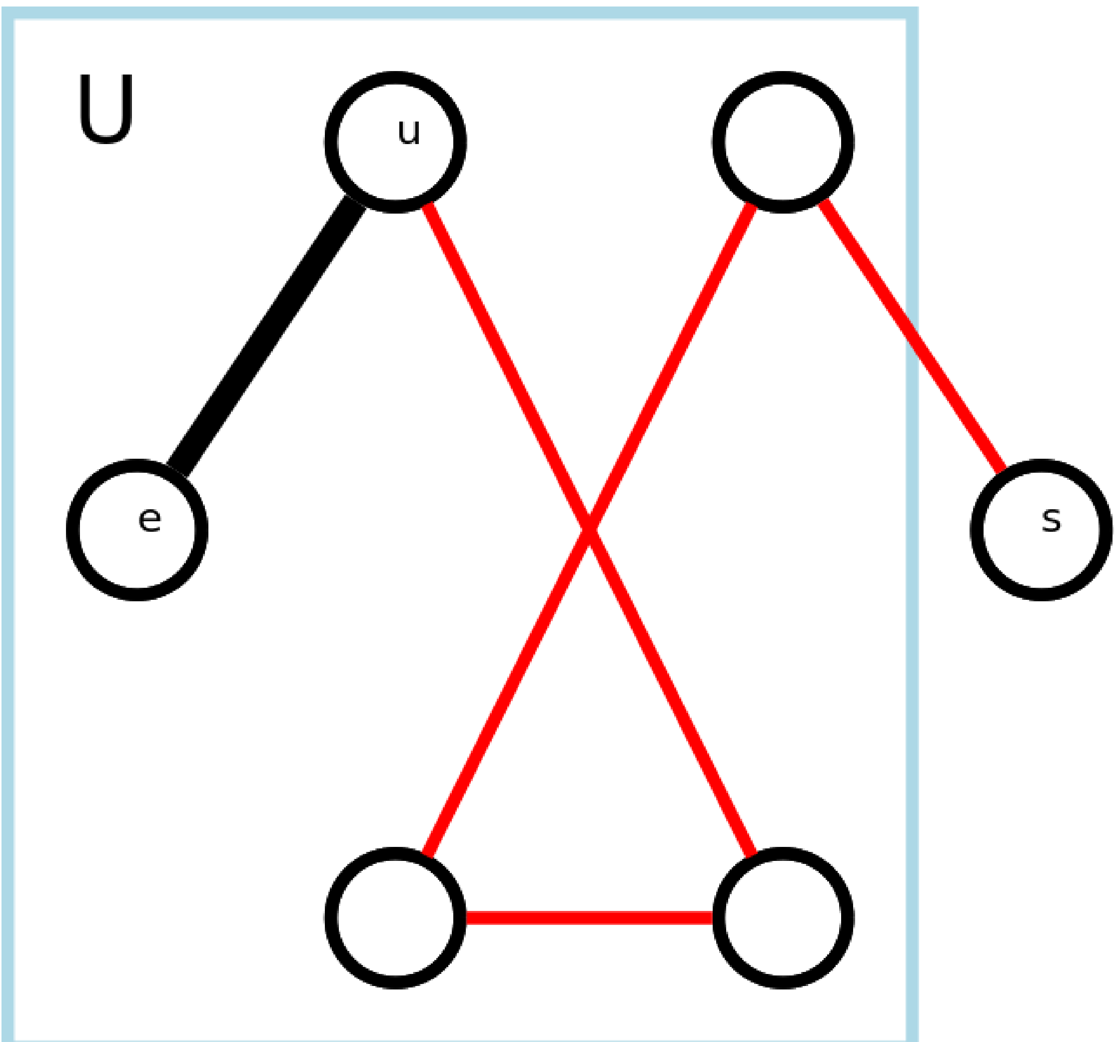
Dynamic TSP

- We first choose an arbitrary starting vertex $s \in V$
- For each nonempty $U \subset V$ and $e \in U$ we compute $OPT[U,e]$, the length of the shortest tour starting in s, visiting all vertices in U and ending in e
- For $|U| = \{e\}$ we trivially set $OPT[U,e] = d(s,e)$
- For $|U| > 1$, $u \in U \setminus \{e\}$, if a tour containing the edge (u,e) is optimal, the tour on $U \setminus \{e\}$ ending in u must be optimal as well
- Thus, for $|U| > 1$, $OPT[U,e]$ is the minimum of $OPT[U \setminus \{e\},u] + d(u,e)$ over all $u \in U \setminus \{e\}$

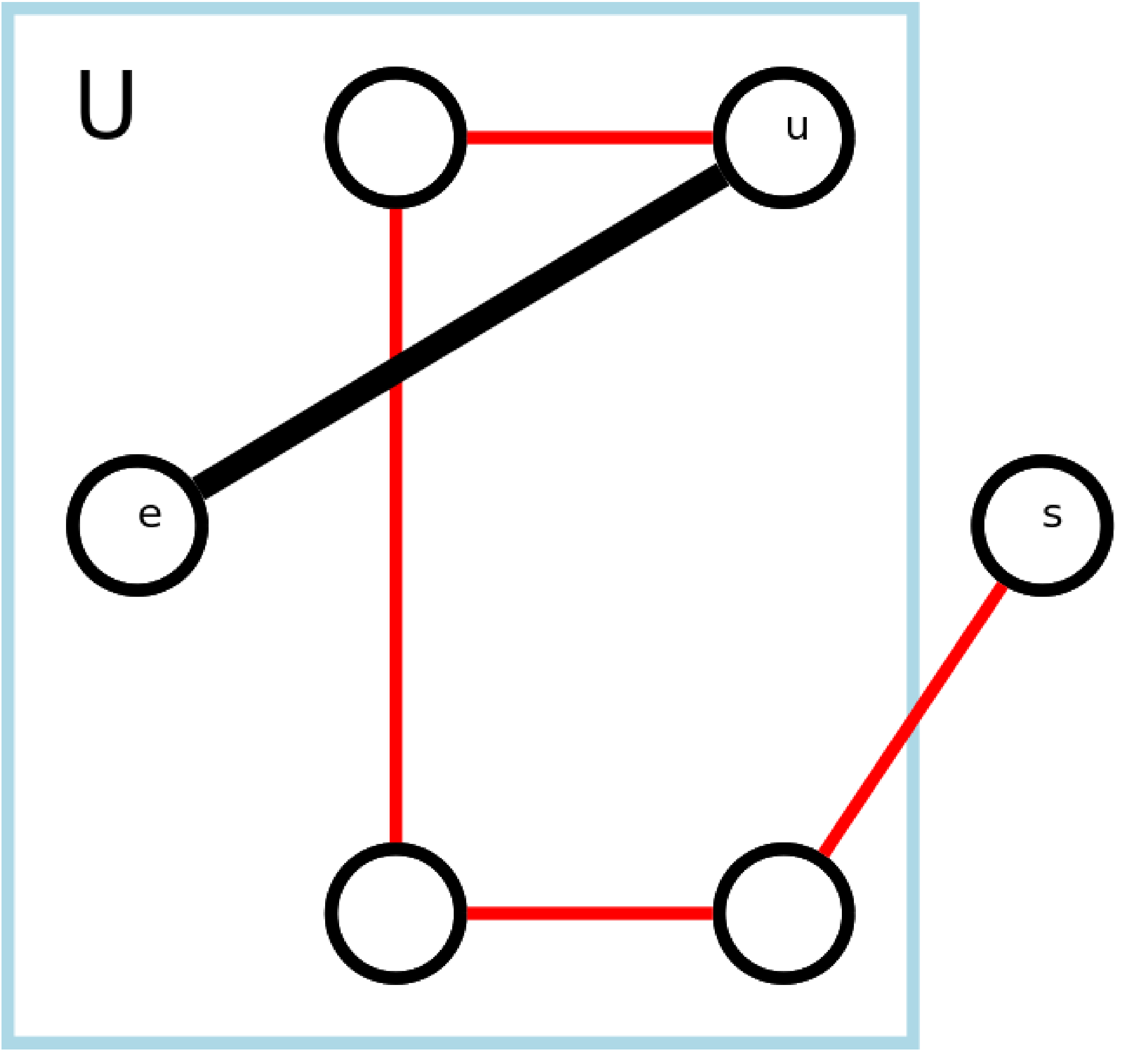
Dynamic TSP



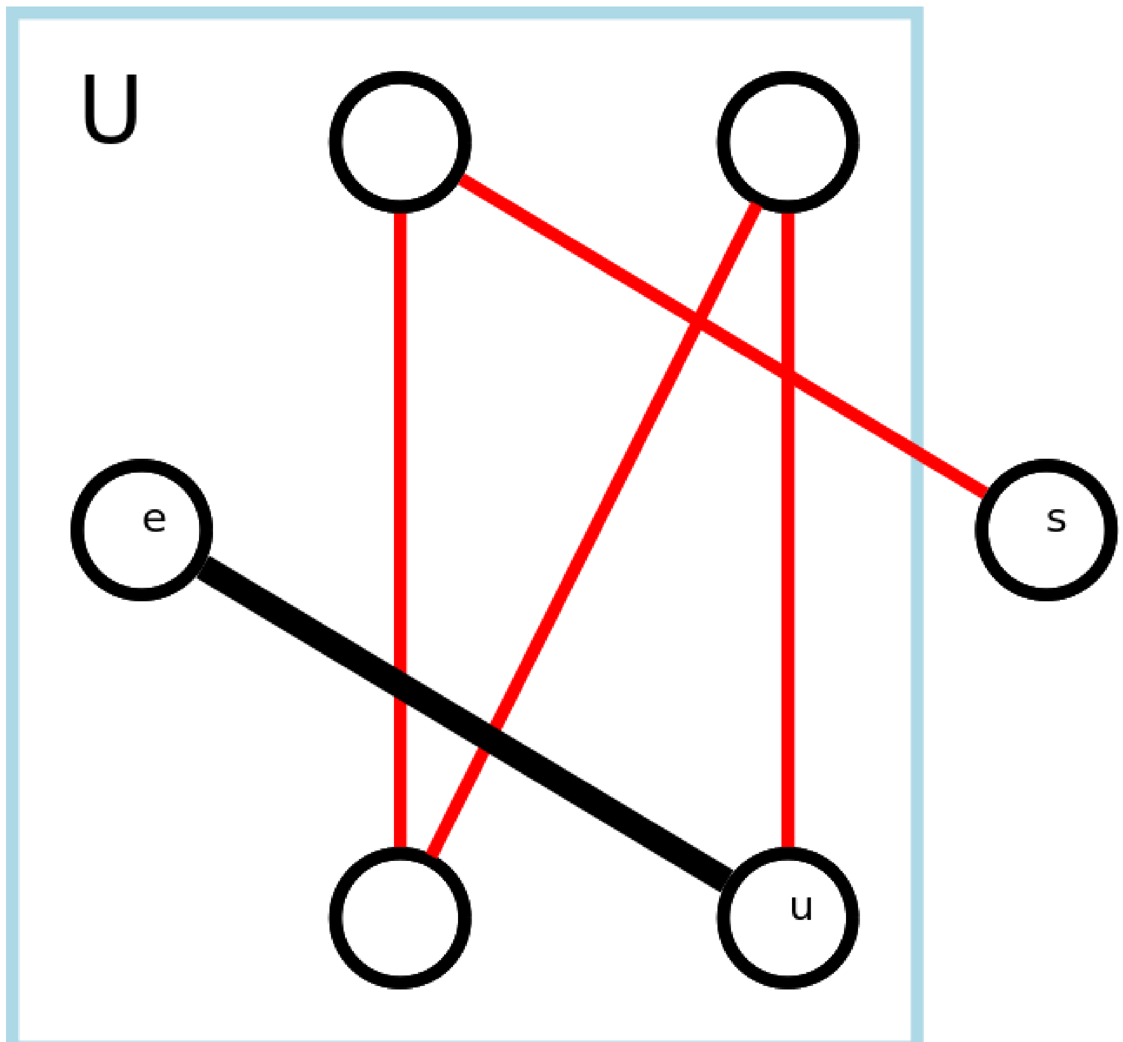
Dynamic TSP



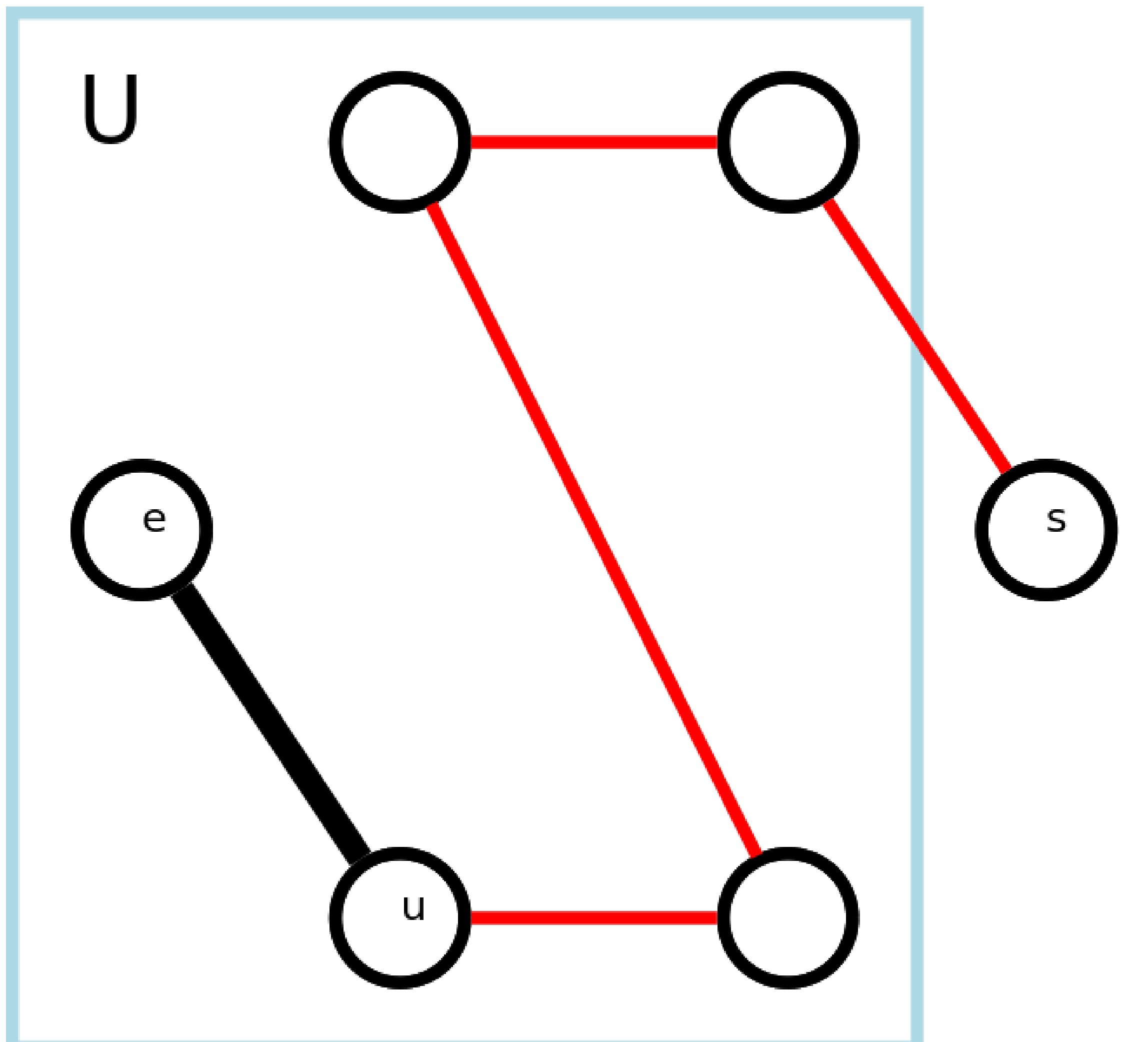
Dynamic TSP



Dynamic TSP



Dynamic TSP



Dynamic TSP

- To compute $OPT[U,e]$, we need the values $OPT[U \setminus \{e\},u]$ for all $u \in U \setminus \{e\}$
  - We compute *OPT* in the order of increasing size of *U* to ensure the values are already computed
  - Computing a single value takes $O(n)$ time
- Finally, $OPT[V,s]$ is the solution to the problem
- The number of subsets is $O(2^n)$ and for each we evaluate the recurrence $O(n)$ times
- Total running time is $O(2^n n^2) = O^*(2^n)$

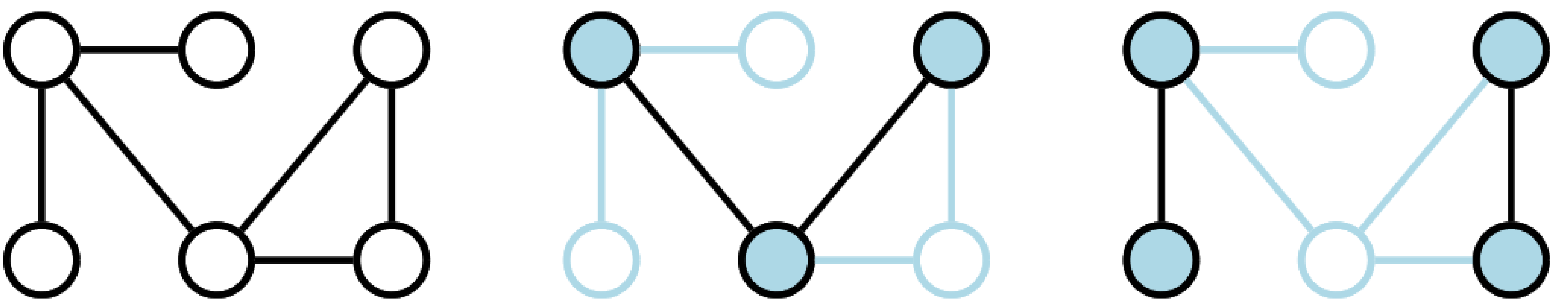
TSP in bounded degree graphs

- Despite its age the dynamic solution is still the best we have
- It's unknown whether a faster algorithm exists
- In some interesting special cases we can can solve TSP in time $O^*((2 - \varepsilon)^n)$ for some $\varepsilon > 0$
- E.g. graphs with bounded maximum degree Δ
  - For cubic graphs ($\Delta = 3$) a branching algorithm solves TSP in time $O^*(1.251^n)$
  - For $\Delta = 4$ we can do it in $O^*(1.733^n)$

TSP in bounded degree graphs

- For $\Delta > 4$ a more recent result bounds the time by $O^*((2 - \varepsilon)^n)$ where $\varepsilon > 0$ depends only on Δ
- Observation: the dynamic algorithm needs to evaluate only tours on *connected sets*
  - $U \subset V$ is a connected set if $G[U]$ is connected
  - Connectedness can be checked in $O(n)$ time
- This yields the running time $O^*(|C|)$ where C is the family of connected sets of the graph
- Analysis is reduced to estimating the size of C

Connected sets



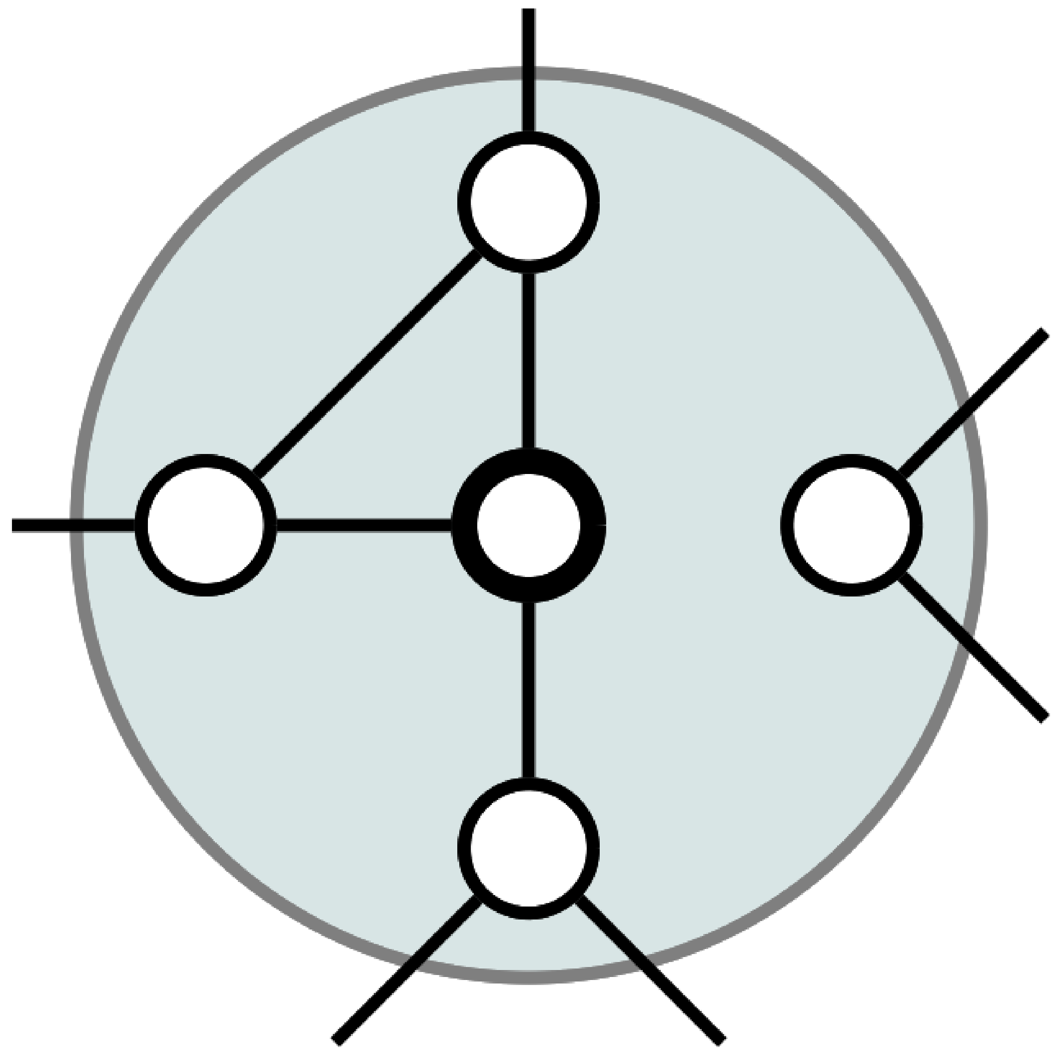
TSP in bounded degree graphs

- For an n-vertex graph of maximum degree Δ we can show that $|C| = O((2^{\Delta + 1} - 1)^{n / (\Delta + 1)})$
- A lemma derived from Shearer's inequality:
  - Let V be a finite set with subsets $A_1, \ldots, A_k$ such that each $v \in V$ is in at least δ subsets
  - Let F be a family of subsets of V
  - Let $F_i = \{S \cap A_i : S \in F\}$ for all $i = 1..k$
  - Then, $|F|^{\delta}$ is at most the product of $|F_i|$ over $i = 1..k$

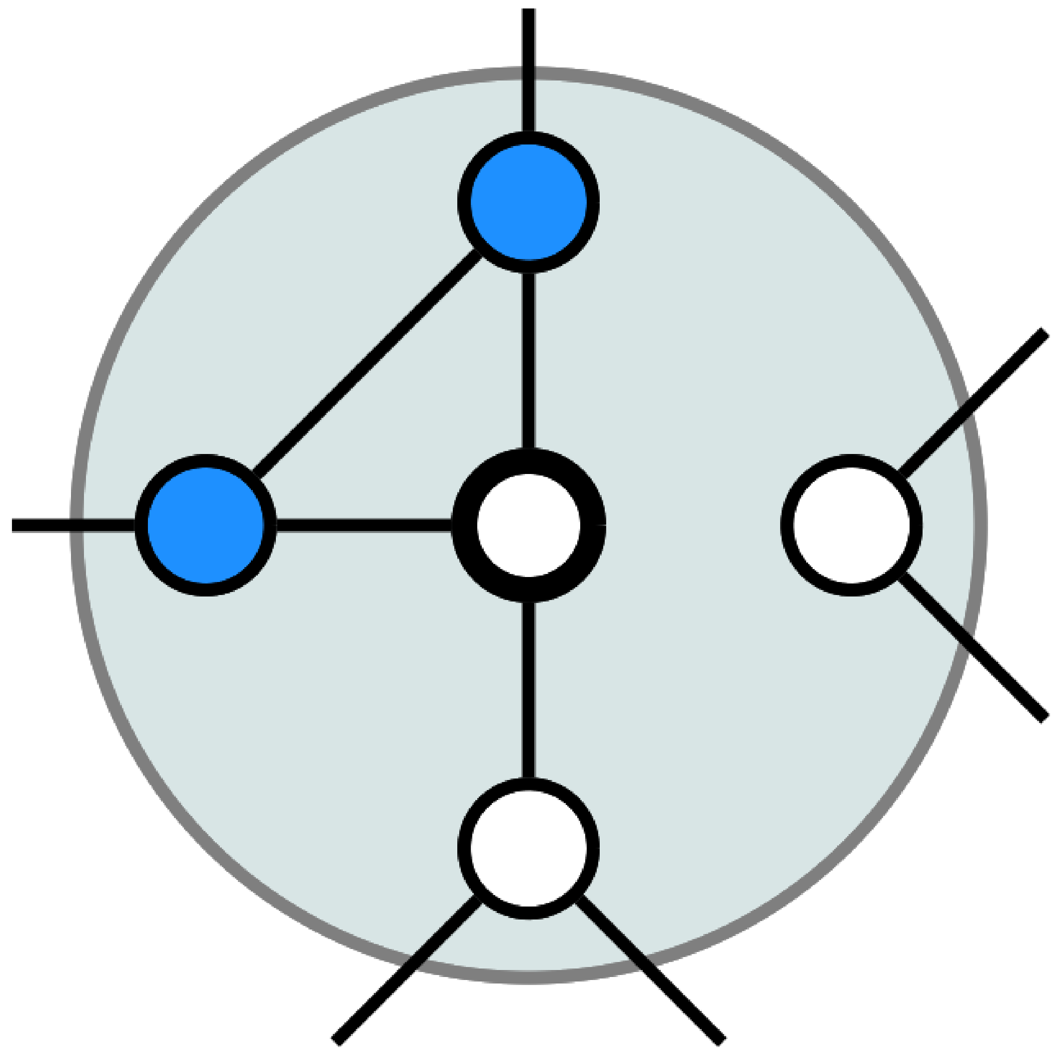
TSP in bounded degree graphs

- For each $v \in V$ we (initially) define A_v as the closed neighborhood of v
- For each $u \in V$ with the degree $d(u) < \Delta$ we add u in $\Delta - d(u)$ sets A_v, chosen arbitrarily
  - Now each $v \in V$ is contained in $\Delta + 1$ subsets
- Define $C' = C \setminus \{\{v\} : v \in V\}$
- And the projections $C_v = \{S \cap A_v : S \in C'\}$ for each $v \in V$

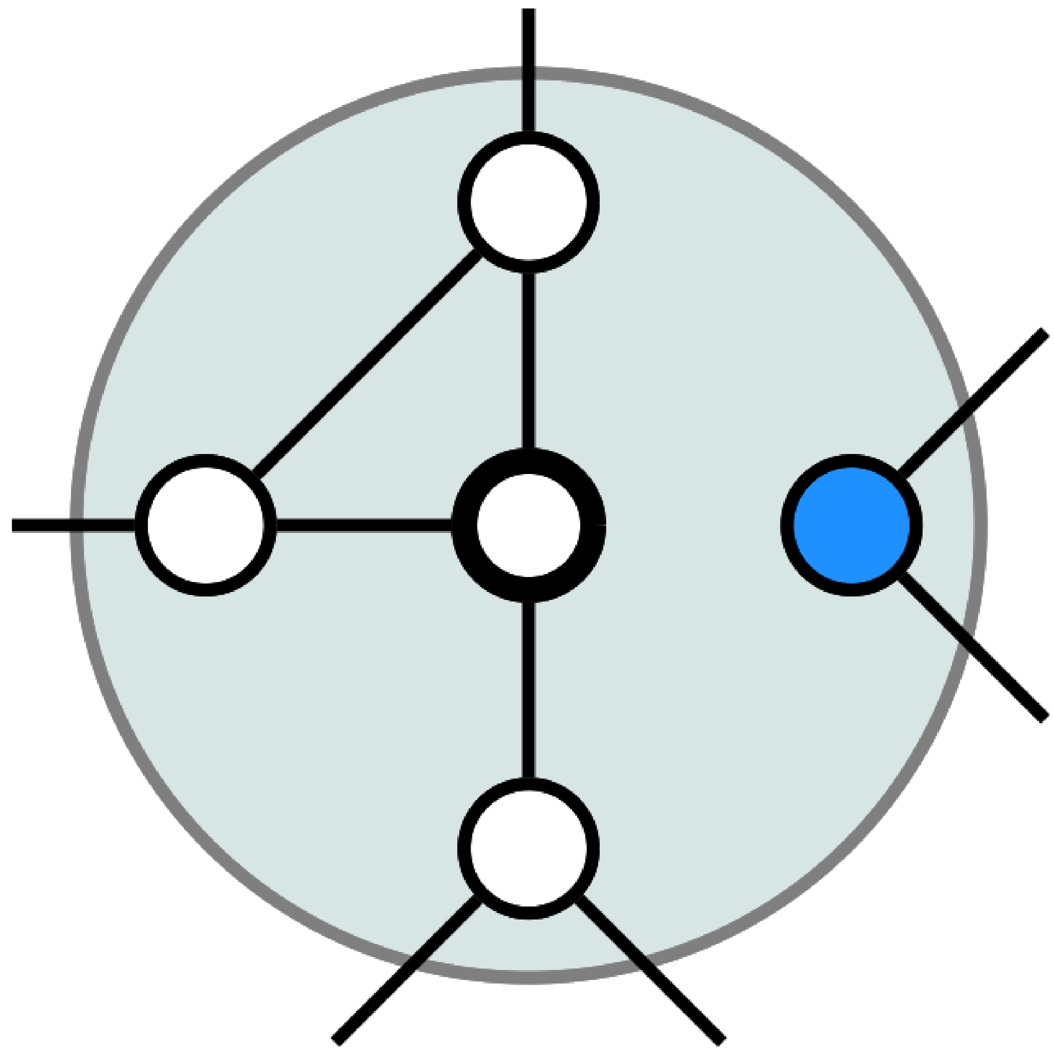
Projection, $\Delta = 3$



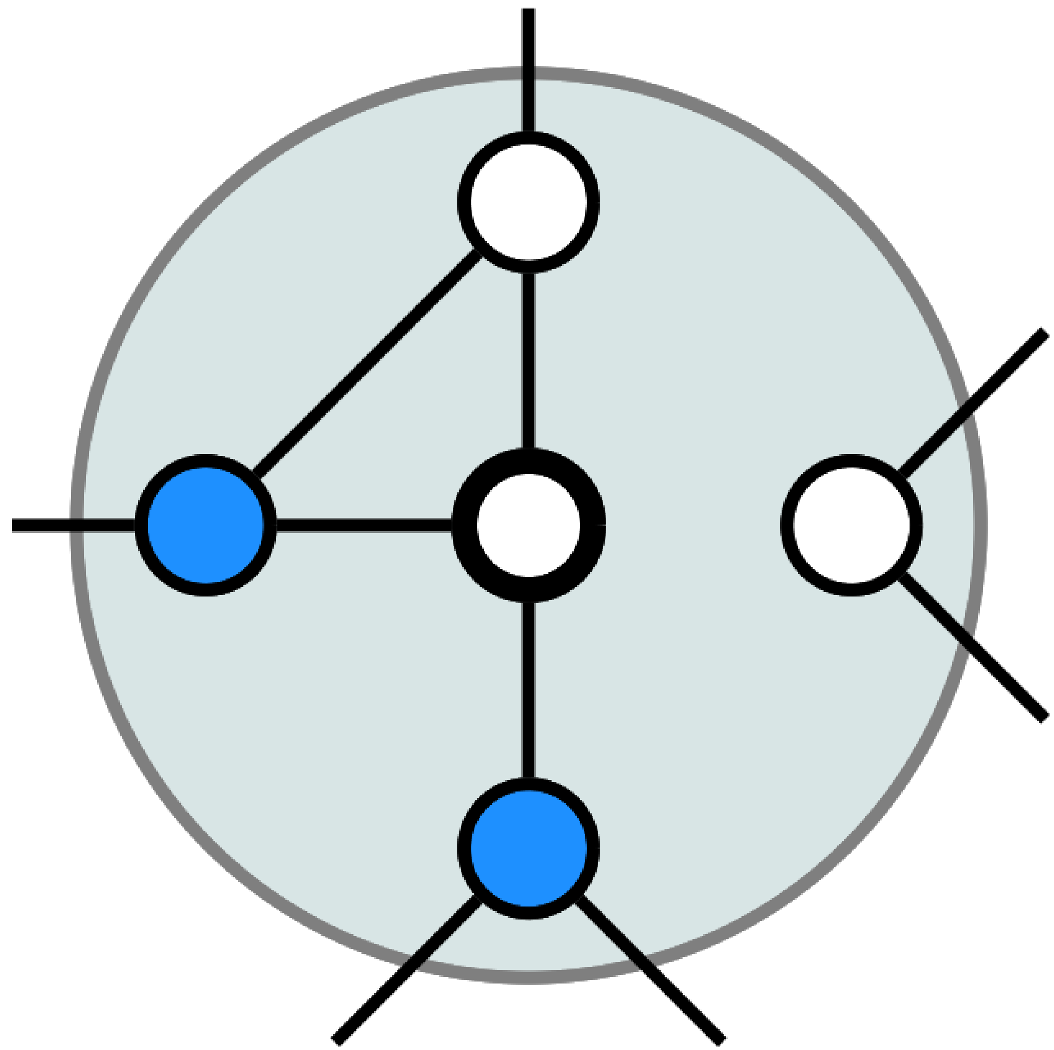
Projection, $\Delta = 3$



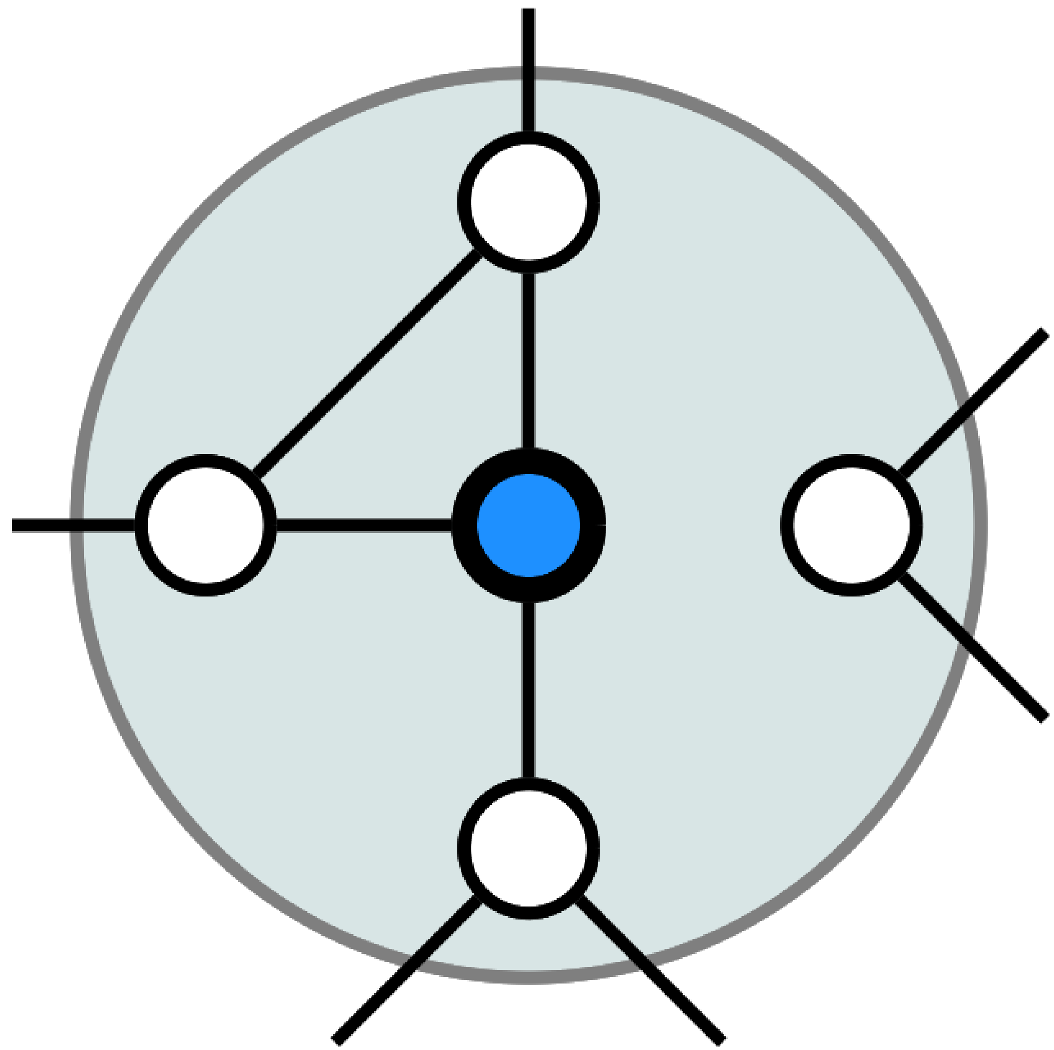
Projection, $\Delta = 3$



Projection, $\Delta = 3$



Projection, $\Delta = 3$



TSP in bounded degree graphs

- Observe that for each $v \in V$ the set C_v does not contain $\{v\}$ since all sets in C' are connected
- Thus, the size of C_v is at most $2^{|A_v|} - 1$
- Shearer: $|C'|^{\Delta + 1}$ is at most the product of $2^{|A_v|} - 1$ over $v \in V$
- With Jensen's inequality we can bound the product (and thus $|C'|^{\Delta + 1}$) by $(2^{\Delta + 1} - 1)^n$
- Thus, the size of $|C'|$ is at most $(2^{\Delta + 1} - 1)^{n / (\Delta + 1)}$
- $|C| = |C'| + n$, yielding the claimed bound

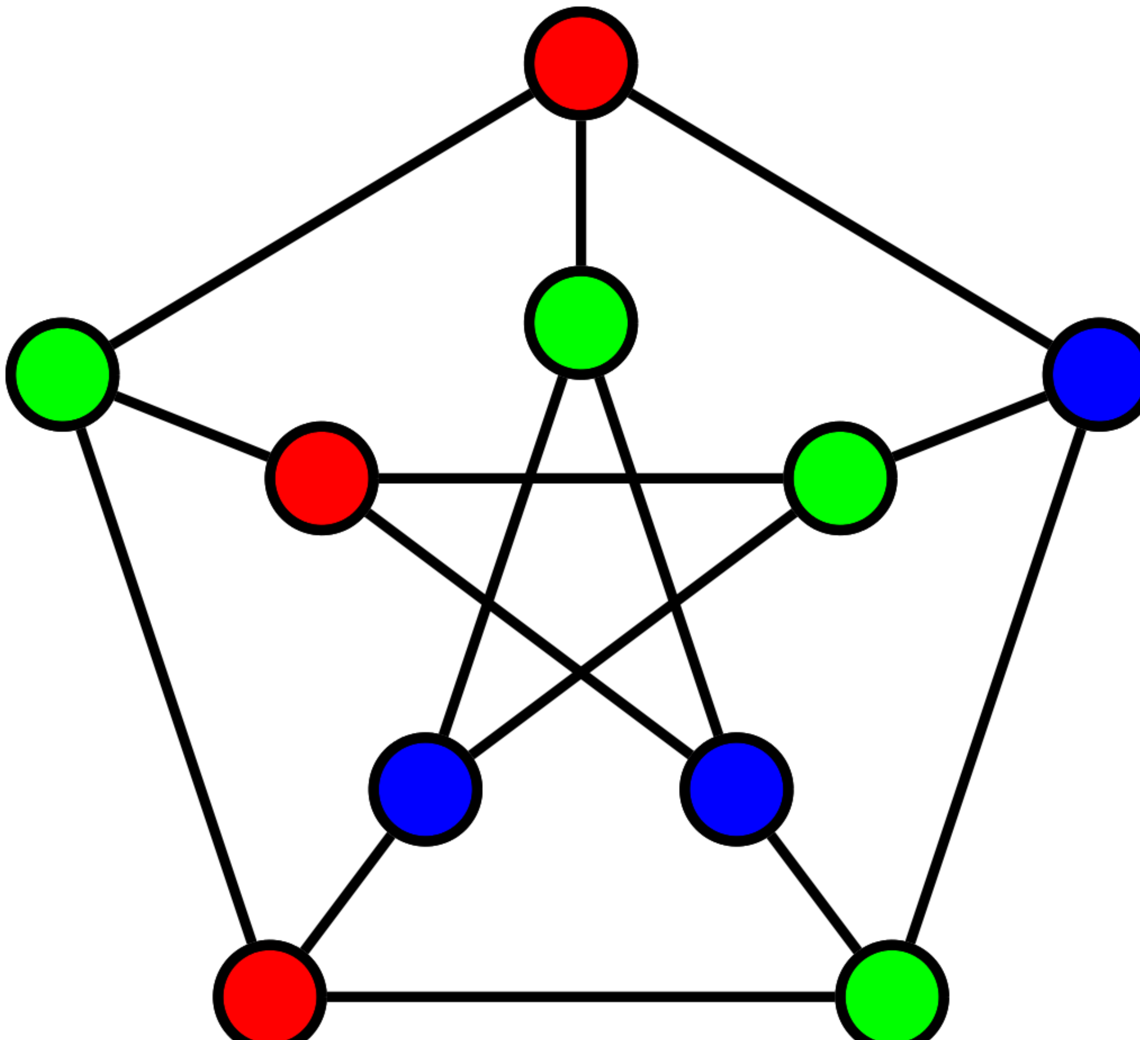
Time-space tradeoff

- In practical applications space complexity is often a greater problem than time
  - Dynamic TSP needs exponential space
- A recursive algorithm that finds similar subtours runs in $O^*(4^n n^{\log n})$ time and polynomial space
  - By switching from recursion to dynamic programming for small subproblems we get a more balanced tradeoff
- Integer-TSP can also be solved in polynomial space and time within a polynomial factor of the number of *connected dominating* sets

Graph coloring

- A k-coloring of an undirected graph $G = (V,E)$ assigns one of k colors to each $v \in V$ such that all adjacent vertices have different colors
- The smallest k with a k-coloring is called the *chromatic number* of G and denoted by $\chi(G)$
- The graph coloring problem asks for either $\chi(G)$ or an *optimal coloring*, using $\chi(G)$ colors
- A partition problem: brute-force search enumerates all partitions of vertices to color classes in $O^*(\chi(G)^n)$ time
- In the worst case $\chi(G) = n$ and the running time is $O^*(n^n)$
- Dynamic programming solves the problem in $O^*(2.4423^n)$

Optimal coloring of Petersen graph



Dynamic graph coloring

- Recall independent sets
  - A subset of vertices $I \subset V$ is an *independent set* if I contains no adjacent vertices
  - I is *maximal* if no proper superset of I is independent
- Observation:
  - A k-coloring is a partition of V into independent sets
  - Each k-coloring can be modified so that at least one set is maximally independent (without increasing k)
  - Consequently, there is an optimal coloring with a maximally independent set

Dynamic graph coloring

- For each $U \subset V$ we find $OPT[U] = \chi(G[U])$, the chromatic number of the subgraph induced by U
- Trivially, $OPT[\emptyset] = 0$
- For $|U| > 0$, an optimal coloring consists of a maximal independent set I and an optimal coloring on the remaining vertices in $G[U \setminus I]$
- Thus, $OPT[U]$ is the minimum of $1 + OPT[U \setminus I]$ over the maximally independent sets I of $G[U]$
- By computing in the order of increasing size of U, we ensure we already have the values $OPT[U \setminus I]$

Dynamic graph coloring

- To compute *OPT*[*U*] we also need to enumerate all maximal independent sets of *G*[*U*]
- This can be done within a polynomial factor of the number of such sets, which for a subgraph of i vertices is at most $3^{i/3}$
- The total number of maximal independent sets over all induced subgraphs of an n-vertex graph is at most $(1 + 3^{1/3})^n = O(2.4423^n)$, and for each we need $n^{O(1)}$ steps, yielding the claimed bound
- Finally, $OPT[V] = \chi(G)$

Conclusion

- Dynamic programming solves a complex problem by breaking it into simpler subproblems
- Subproblems overlap: we compute from simpler to more complex, storing solutions in memory to avoid recomputation
- We can sometimes solve problems with superexponential search space in exponential time, often running on subsets of the problem (e.g. TSP, graph coloring)
- Sometimes we can ignore special subsets and get a more efficient exponential time solution
- Space complexity is often the most restrictive factor