Dynamic Programming

- General method
- Floyd’s all-Pairs Shortest Path
- Traveling salesman problem

General method

- Dynamic programming is an algorithm design method for solving multi-stage decision making problems.
- For some problems greedy methods can be used to generate optimal solutions.
- For many other problems, they do yield optimal solution.
- One way of finding the global optimum is that all decision sequences are enumerated from which the best decision is picked.
- But the time and space requirement may be prohibitive.

Dynamic Programming technique

- In dynamic programming a collection of decision sequences are generated.
- Essential difference from greedy method
- In Greedy method only one sequence of decisions is generated. e.g.
  - Knapsack: We chose a sequence of objects fractions such that it optimizes the objective function.
  - Shortest Path: We chose a sequence of vertices such that it minimizes the path length.
- Solve smaller subproblems of the same type.

- Solution to problem is represented as a recurrence relation
- which links the solutions to the smaller problems into a solution to the whole problem.
- It employs principle of optimality: *An optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must produce an optimal decision sequence with regard to the state resulting from the first decision.*
- It reduces the number of possible solutions to be checked.
  - By eliminating redundant information (overlapping problems)
  - Removing cases that cannot be optimal

Floyd's all pairs shortest path

String edit

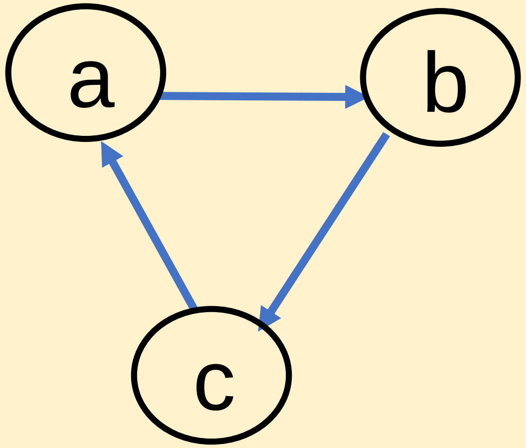
Dynamic Programming

Warshall transitive closure

Traveling salesman

The transitive closure

- Given a directed graph with n vertices $G\ (V, E)$



- The transitive closure is denoted as an Boolean matrix

  $R=\{r_{ij}\}$

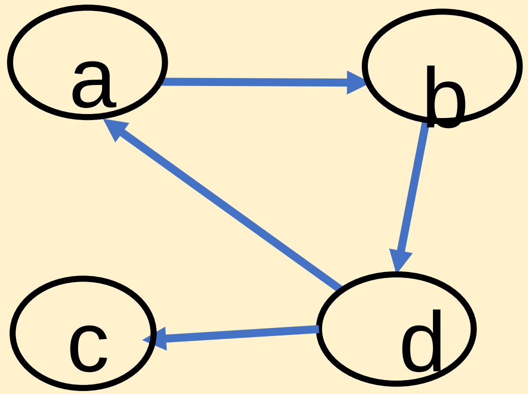
- in which the row i and column j is 1 (one) if there exists a non-trivial directed path

$$G = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Warshall's Transitive Closure Algorithm

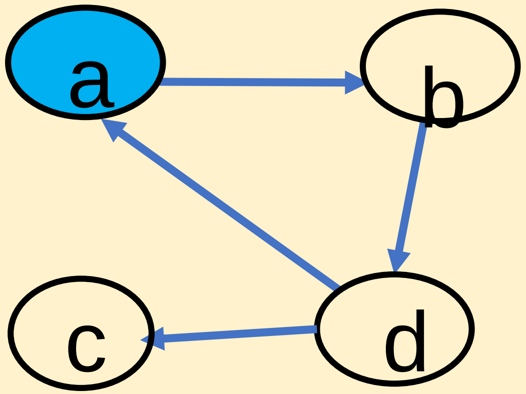
- Recursively create $R^{0}, \cdots R^{n}$ by adding intermediate vertices as in Floyd algorithm
- Generation of from R^{k-1} has the following rules:
  - If an element r_{ij} is 1 in R^{k-1} ,it remains 1 in R^k
  - If an element r_{ij} is 0 in R^{k-1} , it has to be changed to 1 in R^k, if and only if the element in row i and column k and the elements in column j and row k are both 1s in R^{k-1} .



$$R^0 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Warshall's Transitive Closure Algorithm

- If an element r_{ij} is 0 in R^{k-1}, it has to be changed to 1 in R^k, if and only if the element in row i and column k and the elements in column j and row k are both 1s in R^{k-1}.



$$R^0 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

$$R^1 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

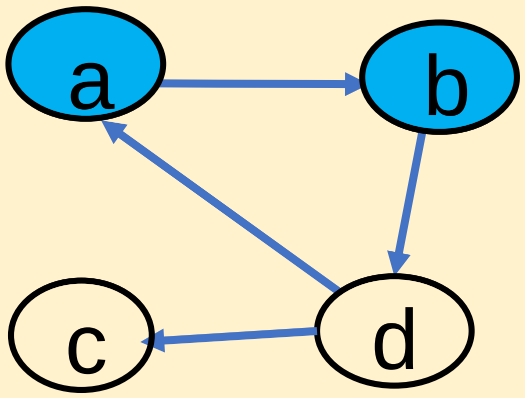
$d \to a \to b$

$a \to b \to d$

$b \to d \to a$

$b \to d \to c$

- If an element r_{ij} is 0 in R^{k-1}, it has to be changed to 1 in R^k, if and only if the element in row i and column k and the elements in column j and row k are both 1s in R^{k-1}.



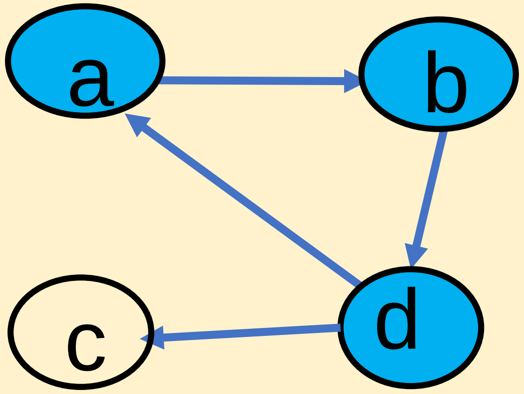
$$R^1 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

$d \to a \to b$
$a \to b \to d$
$b \to d \to a$

$a \to b \to d \to a$
$b \to d \to a \to b$
$d \to a \to b \to d$
$a \to b \to d \to c$

$$R^2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

- If an element r_{ij} is 0 in R^{k-1}, it has to be changed to 1 in R^k, if and only if the element in row i and column k and the elements in column j and row k are both 1s in R^{k-1}.



$$R^2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$a \to b \to d \to a$

$b \to d \to a \to b$

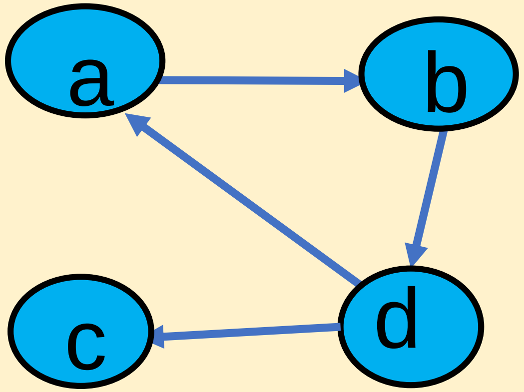
$d \to a \to b \to d$

$a \to b \to d \to c$

$$R^3 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

In this case, no element update

- If an element r_{ij} is 0 in R^{k-1}, it has to be changed to 1 in R^k, if and only if the element in row i and column k and the elements in column j and row k are both 1s in R^{k-1}.



$$R^3 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \qquad R^4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

In this case, no element update

- $R = R^4$ is what we are looking for.

- *Input: Graph G of size n*
- *Output: transitive closure matrix*
- *Copy adjacent matrix G to R*
- *Elements of each matrix R^k can be computed from its immediate predecessor R^{k-1} by recurrence*

```
Algorithm Warshall( Graph G){
    for(i= 1, i <=n, i+ +)
       for(j= 1, j <=n, j+ +)
          D[i, j] =W[i, j];
    for(k= 1, k <=n, k+ +)
          f or(i= 1, i <=n, i+ +)
             for(j= 1;j <=n;j+ +){
            R[i, j] =R[i, j] or (R[i, k]and R[k, j]
   }
   Return R
}
```

Elements of each matrix R^k can be computed from its immediate predecessor R^{k-1} with the following recurrence:

$$r_{i,j}^{0} = G_{i,j}^{0}, \quad r_{i,j}^{k} = r_{i,j}^{k-1} \text{ OR } (r_{i,k}^{k-1} \text{ and } r_{k,j}^{k-1})$$

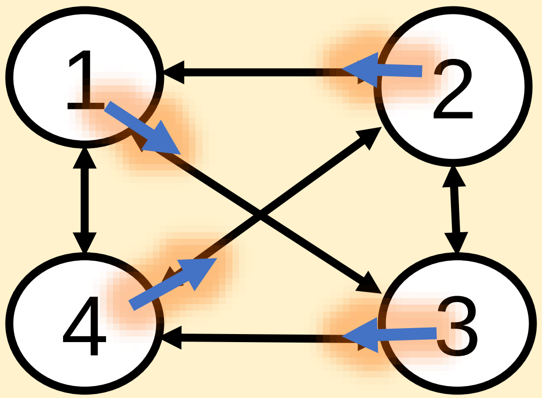
Travelling Salesman Problem (TSP)

- The Traveling Salesman Problem is one of the most intensively studied problems in computational mathematics.
- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?
- path that passes through every vertex exactly once and returns to the starting vertex is an Hamiltonian Circuit.
- TSP’s “optimal tour” is an Hamiltonian Circuit of minimum weight.

- The Traveling Salesman Problem is one of the most intensively studied problems in computational mathematics.
- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?
- path that passes through every vertex exactly once and returns to the starting vertex is an Hamiltonian Circuit.
- TSP's "optimal tour" is an Hamiltonian Circuit of minimum weight.
- TSP is an NP-Complete Problem
  - No polynomial time algorithm exists.
  - Exhaustive searching is the only guaranteed way to find optimal tour.

Example

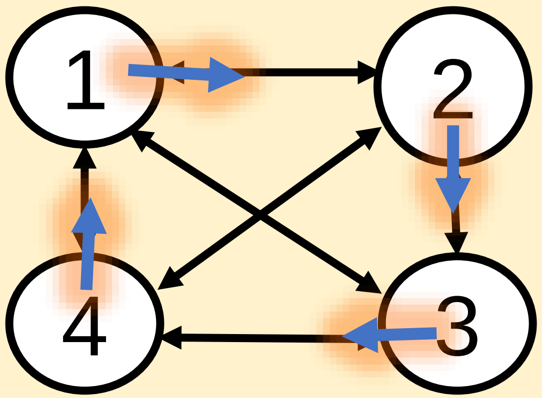
$$c = \begin{bmatrix} 0 & 8 & \color{red}{6} & 10 \\ \color{red}{4} & 0 & 8 & 7 \\ 6 & 5 & 0 & \color{red}{5} \\ 7 & \color{red}{6} & 4 & 0 \end{bmatrix}$$



- By enumeration there are $n! = 4! = 24$ possible tours

1,2,3,4	*8+8+5+7*	*=28*
1,2,4,3	*8+7+4+6*	*=25*
1,3,2,4	*6+5+7+7*	*=25*
1,3,4,2	*6+5+6+4*	*=21*
1,4,2,3	*10+6+8+6*	*=30*
1,4,3,2	*10+4+5+4*	*=23*

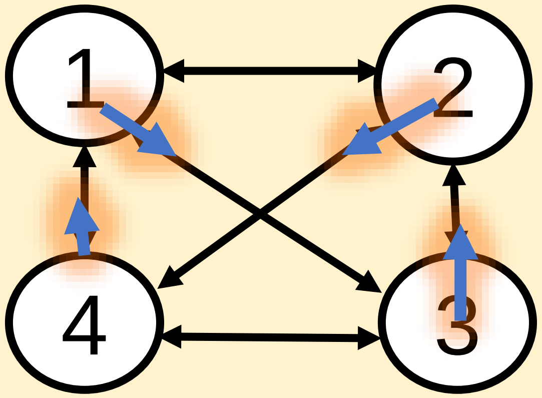
$$c = \begin{bmatrix} 0 & 8 & 6 & 10 \\ 4 & 0 & 8 & 7 \\ 6 & 5 & 0 & 5 \\ 7 & 6 & 4 & 0 \end{bmatrix}$$



- By enumeration there are *n! =4! =24* possible tours

2,1,3,4	*4+6+5+6*	*=21*
2,1,4,3	*4+10+4+5*	*=23*
2,3,1,4	*8+6+10+6*	*=30*
2,3,4,1	*8+5+7+8*	*=28*
2,4,1,3	*7+7+6+5*	*=25*
2,4,3,1	*7+4+6+8*	*=25*

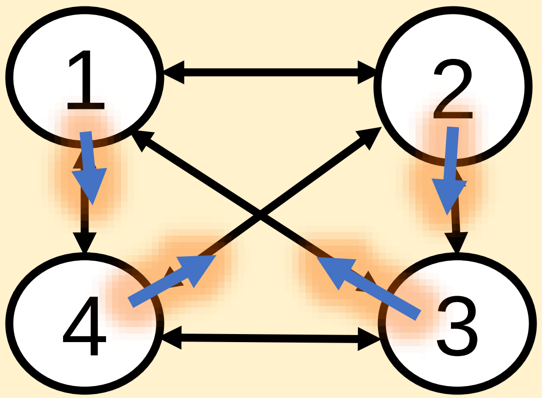
$$c = \begin{bmatrix} 0 & 8 & 6 & 10 \\ 4 & 0 & 8 & 7 \\ 6 & 5 & 0 & 5 \\ 7 & 6 & 4 & 0 \end{bmatrix}$$



- By enumeration there are *n! =4! =24* possible tours

3,1,2,4	*6+8+7+4*	*=25*
3,1,4,2	*6+10+6+8*	*=30*
3,2,1,4	*5+4+10+4*	*=23*
3,2,4,1	*5+7+7+6*	*=25*
3,4,1,2	*5+7+8+8*	*=28*
3,4,2,1	*5+6+4+6*	*=21*

$$c = \begin{bmatrix} 0 & 8 & 6 & \color{red}{10} \\ 4 & 0 & \color{red}{8} & 7 \\ \color{red}{6} & 5 & 0 & 5 \\ 7 & \color{red}{6} & 4 & 0 \end{bmatrix}$$



- By enumeration there are $n! = 4! = 24$ possible tours

4,1,2,3	*7+8+8+5*	*=28*
4,1,3,2	*7+6+5+7*	*=25*
4,2,1,3	*6+4+6+5*	*=21*
4,2,3,1	*6+8+6+10*	*=30*
4,3,1,2	*4+6+8+7*	*=25*
4,3,2,1	*4+5+4+10*	*=23*

Optimal Tour by Dynamic Programming

- In 13 steps (instead of 24) we will compute the optimal tour for the 4 city example.
- Eliminates repeat calculations of implausible tours.
- We do this by constructing a recurrence equation
- which captures the principle of optimality
- And then unwind the recurrence equation 13 times

Principle of optimality: An optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must produce an optimal decision sequence with regard to the state resulting from the first decision.

- Assume we start at vertex "1" in G(V, E)
- Every tour consists of
  - An edge $<1, k>$ for some k from the set of vertices k in the set $V-\{1\}=\{2,3,4\}$
  - And a path from vertex k to the starting vertex 1.
- If the tour is optimal, path from k back to 1 should be optimal
- Hence principle of optimality holds
- Let $g(i, S)$ be the length of a shortest path starting at vertex i, going through all vertices in a path S, terminating at vertex 1.
- From principle of optimality, the optimal tour can be specified as:

$$g(1, V-\{1\}) = \min_{2\leq k\leq n}\{c[1,k] + g(k, V-\{1, k\})\}$$

- i.e.

$$g(1, V-\{1\}) = \min_{2\le k\le n} \{c[1,k] + g(k, V-\{1, k\})\}$$

- We have to solve for

$$g(1, \{2,3,4\}) = \min\{c(1,2) + g(2,\{3,4\}), c(1,3) + g(3,\{2,4\}), c(1,4) + g(4,\{2,3\})\}$$

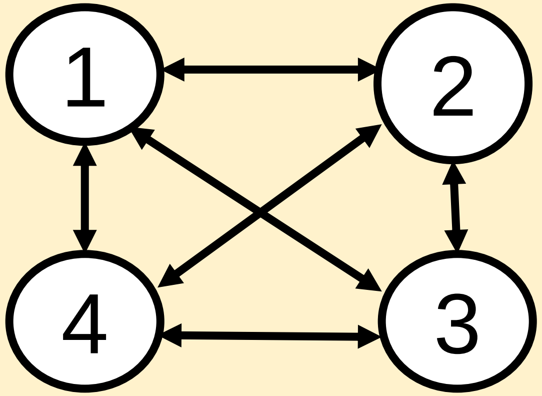
- Generalise the above equation to

$$g(i, S) = \min_{j\in s} \{c(i, j) + g(j, S-\{j\}) \}$$

for $i \ne S$.

Example $g(i, S) = \min_{j \in s} \{c(i, j) + g(j, S-\{j\})\}$ for $i \neq S$.

$$c = \begin{bmatrix} 0 & 8 & 6 & 10 \\ 4 & 0 & 8 & 7 \\ 6 & 5 & 0 & 5 \\ 7 & 6 & 4 & 0 \end{bmatrix}$$



- Using generalized equation with $|S| = 0$:

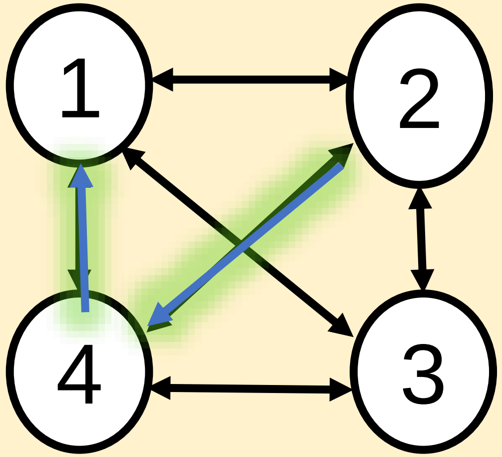
  $g(2,\{\ \}) = c(2,1) = 4$

  $g(3,\{\}) = c(3,1) = 6$

  $g(4,\{\}) = c(4,1) = 7$

$g(i,\{\})$ is path length from i to 1 directly

$g(i, S) = \min_{j \in s} \{c(i, j) + g(j, S-\{j\})\}$ for $i \neq S$.



➢ Using generalized equation with $|S| = 1$:

$g(2,\{3\}) = c(2,3) + g(3,\{\}) = c(2,3) + c(3,1)$
$= 8+6 = 14$

$\boldsymbol{g(2,\{4\}) = c(2,4) + g(4,\{\}) = c(2,4) + c(4,1)}$
$\boldsymbol{= 7+7 = 14}$

$g(3,\{2\}) = c(3,2) + g(2,\{\}) = c(3,2) + c(2,1)$
$= 5+4 = 9$

$g(3,\{4\}) = c(3,4) + g(4,\{\}) = c(3,4) + c(4,1)$
$= 5+7 = 12$

$g(4,\{2\}) = c(4,2) + g(2,\{\}) = c(4,2) + c(2,1)$
$= 6+4 = 10$

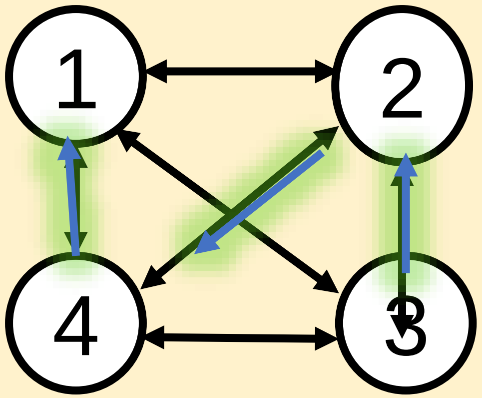
$g(4,\{3\}) = c(4,3) + g(4,\{\}) = c(4,3) + c(3,1) = 4+6 = 10$

$$c = \begin{bmatrix} 0 & 8 & 6 & 10 \\ 4 & 0 & 8 & 7 \\ 6 & 5 & 0 & 5 \\ 7 & 6 & 4 & 0 \end{bmatrix}$$

$g(i,\{j\})$ is path length from i to 1 via j

$$g(i, S) = \min_{j \in s} \{c(i, j) + g(j, S - \{j\})\} \qquad \text{for } i \neq S.$$

$$c = \begin{bmatrix} 0 & 8 & 6 & 10 \\ 4 & 0 & 8 & 7 \\ 6 & 5 & 0 & 5 \\ 7 & 6 & 4 & 0 \end{bmatrix}$$



- Using generalized equation with $|S| = 2$:

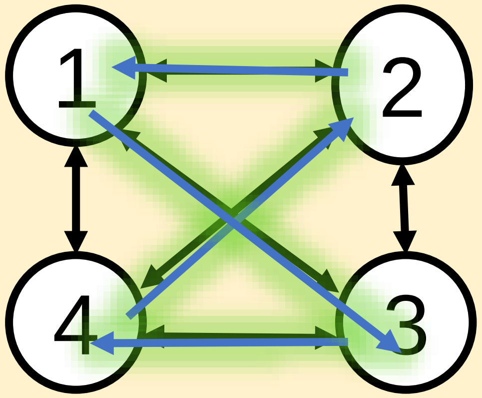
$g(2,\{3,4\}) = \min(c(2,3)+g(3,\{4\}), c(2,4)+g(4,\{3\})) = \min(20,17)$
$= 17$

$g(3,\{2,4\}) = \min(\mathbf{c(3,2)+g(2,\{4\})}, c(3,4)+g(4,\{2\})) = \min(19,25)$
$= \mathbf{19}$

$g(4,\{2,3\}) = \min(c(4,2)+g(2,\{3\}), c(4,3)+g(3,\{2\})) = \min(20,13)$
$= 13$

$g(i, S) = \min_{j \in s} \{c(i, j) + g(j, S - \{j\})\}$ for $i \neq S$.

$$c = \begin{bmatrix} 0 & 8 & 6 & 10 \\ 4 & 0 & 8 & 7 \\ 6 & 5 & 0 & 5 \\ 7 & 6 & 4 & 0 \end{bmatrix}$$



- Finally :

$$g(1,\{2,3,4\}) = \min\{c(1,2) + g(2,\{3,4\}), \mathbf{c(1,3) + g(3,\{2,4\})}, c(1,4) + g(4,\{2,3\})\}$$

$$= \min(25, \mathbf{21}, 23) = 21$$

- Trace back to get tour (1,3), (3,2), (2,4),(4,1)

Or 1−>3−>2−>4−>1

End of term revision

Exam structure

- Answer Three from Five for CS2AO17
- Topics covered in exam
  - Additional data structures
    - Tree, Heap, Graph, tree and graph traversals
  - **Algorithms**
    - Divide and Conquer
    - **The greed algorithm***
    - **Dynamic programming***

- ***This Revision**

The Greedy Method

The greedy method suggests constructing a solution through a sequence of steps.

On each step the choice made must be:

- **feasible** ie., it has to satisfy the problem constraints.
- **(locally) optimal** ie. it has to be the best choice among all feasible choices.

irrevocable ie. the choice once made cannot be changed on subsequent steps of the algorithm.

Works well when applied to problems with **greedy-choice** property:

“a globally optimal solution may be obtained by from a series of local improvements from some starting solution set.”

Control abstraction

Define solutions of problems by Control abstraction

- Flow of control is un-ambiguous
- primary operations are undefined (Select, Feasible , Union)

```
Algorithm Greedy (A:set; n:integer){
   MakeEmpty(solution);
   for(i= 2;i <=n;i+ +){
   x=Select(A);
   if Feasible(solution, x) then
   solution=Union(solution;{x})
   }
   return solution
}
```

Control abstraction is crucial to the design and maintenance of any large software system, ignoring details of implementation focus attention on details of greater importance.

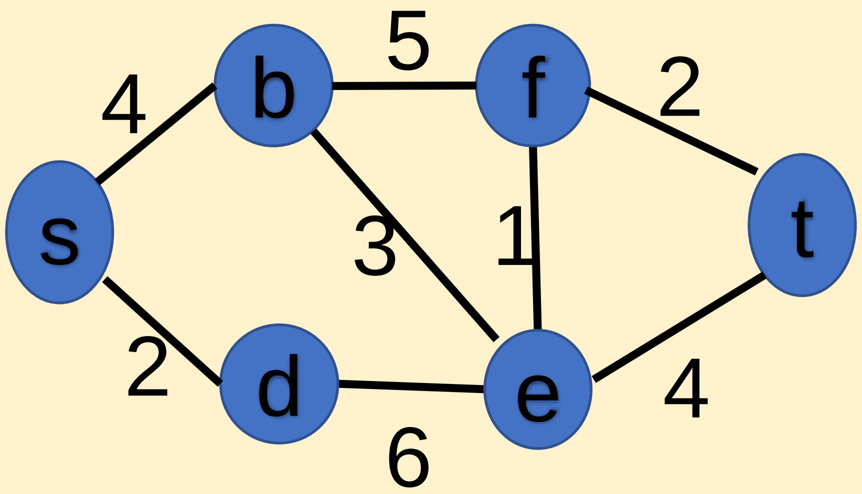
Weighted Graph — Shortest Path

Given a weighted graph, find the shortest directed path from s to t

Cost of path = sum of edge costs in the path

Definition: if $P = e_1 e_2 e_3 \cdots e_k$ are edges connecting source (s) to a destination (t), the length/weight of a path is the sum of the weights of its edges, i.e,

$$w(P) = \sum_{i=1}^{k} e_i$$



Greedy approach

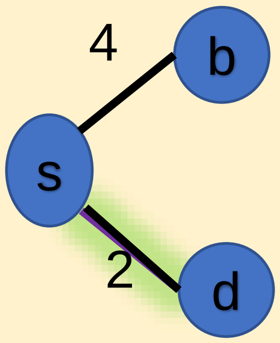
Generate the paths starting from some vertex according to increasing order of path length

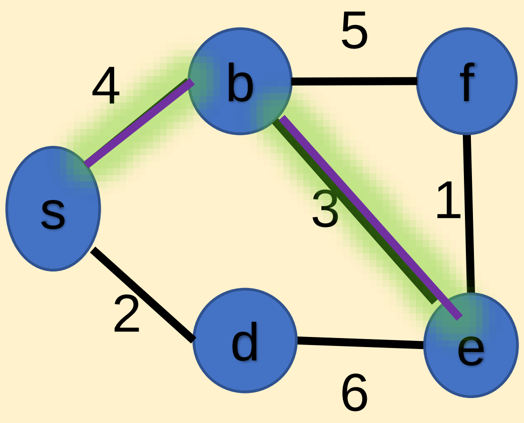
Feasible

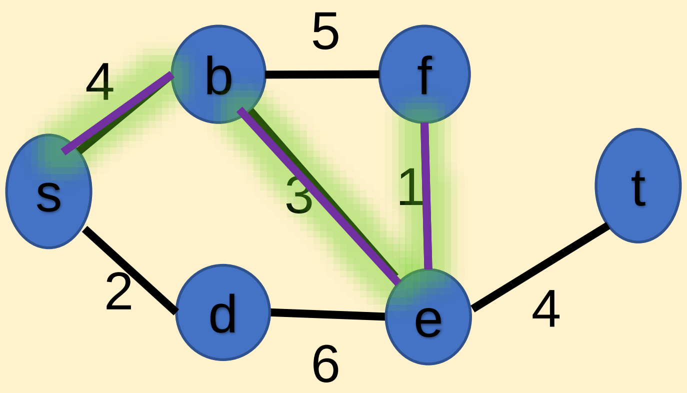
– Every sub-path to a particular vertex is a feasible solution

Optimal

– sum of the lengths of all paths so far generated should be minimal





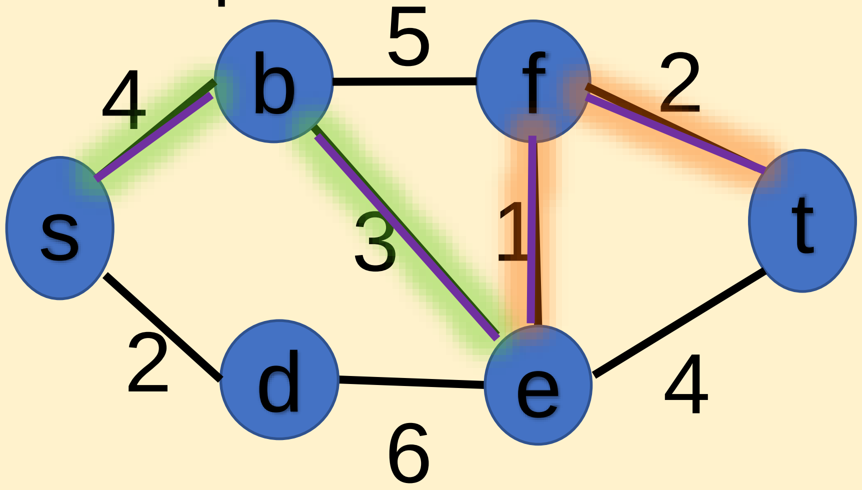


Dijkstra's Shortest Path Algorithm

Dijkstra's algorithm is based on the property that

If a shortest path from s to t goes through vertex e

- then the sub-path from s to e is a shortest path from s to e.
- and the sub-path from e to t is a shortest path from e to t



Kruskal’s MST Algorithm

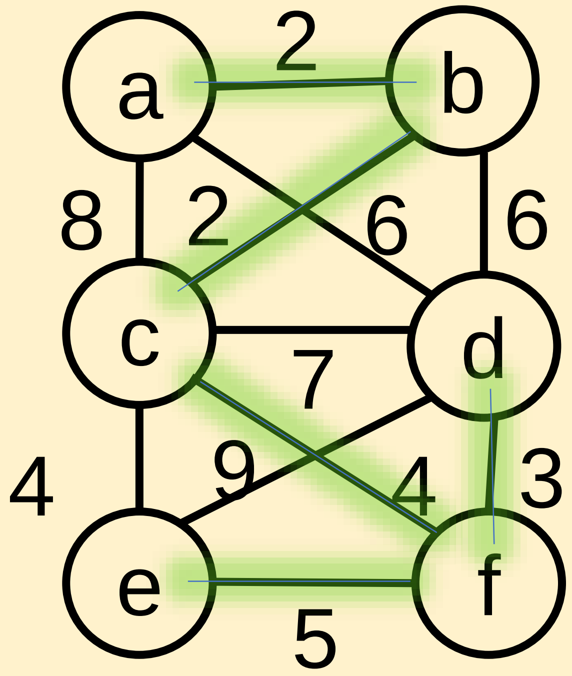
Kruskal’s Greedy Strategy: “greedily” expand a sequence of subgraphs into a acyclic “bigger” sub-graph that is a tree.

Kruskal’s Algorithm:

- Sort the edges in increasing order
- Start with an empty sub-graph
- Add the next edge on the list to the current graph
- If an inclusion results in a cycle discard the edge

Kruskal's -- example

E_1 = (a,b) , T = {(a,b)}, cost=2
E_2 = (b,c), T = {(a,b), (b,c)}, cost=4
E_3 = (d,f), T = {(a,b), (b,c) (d,f)}, cost=7
E_4 = (c,f), T= {(a,b), (b,c) (d,f) (c,f)}, cost=11
E_5 = (c,e), T= {(a,b), (b,c) (d,f) (c,f) (c,e)}, cost=15
E_6 = (e,f), reject
E_7 = (b,d), reject
E_8 = (a,d), reject
E_9 = (c,d), reject
E_{10} = (a,c), reject
E_{11}= (e,d), reject



Dynamic Programming

Essential difference from greedy method

- In Greedy method only one sequence of decisions is generated. e.g.
  - Knapsack: We chose a sequence of objects fractions such that it optimizes the objective function.
  - Shortest Path: We chose a sequence of vertices such that it minimizes the path length.
- In dynamic programming a collection of decision sequences are generated and an optimal one chosen from the collection.
  - Local information may not be adequate to generate global optimum

Dynamic Programming technique

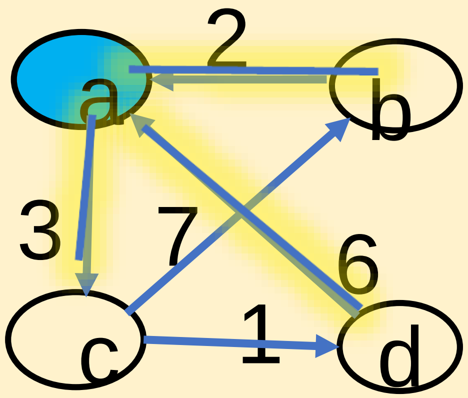
Solve smaller sub-problems of the same type

◆ Solution to problem is represented as a recurrence relation which links the solutions to the smaller problems into a solution to the whole problem.

E.g.

- $d_{i,j}^{0} = W_{i,j}^{0}, \quad d_{i,j}^{k} = \min\{d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1}\}$,

Floyd's All-Pairs Shortest Path



Introduction of an intermediate vertex ***a*** gives two detours

- We may regard the construction of a shortest (i, j) path as first requiring a decision as which is the shortest past path for k intermediate vertices.
- Once this decision has been made, we need to find two shortest paths

  one from i to k

  other from k to j

$$D^0 = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix}$$

$b \to a \to c$

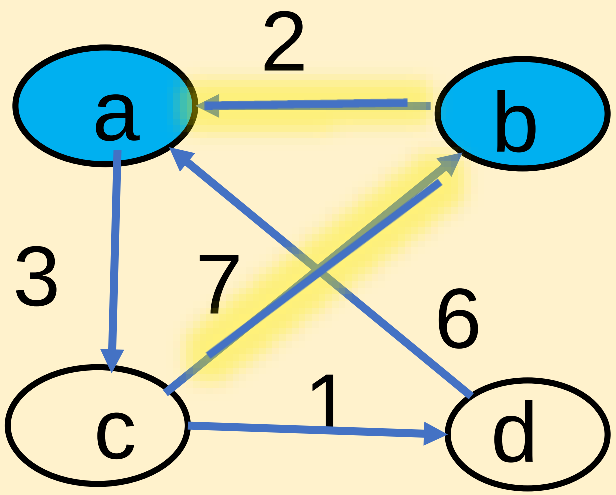
$d \to a \to c$

$$D^1 = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

New possibilities from b to c and from d to c

D(b,c) = min (∞ , D(b,a) + D (a,c)) = min (∞ , 2 + 3) = 5

Introduction of an intermediate vertex ***b*** gives one detours

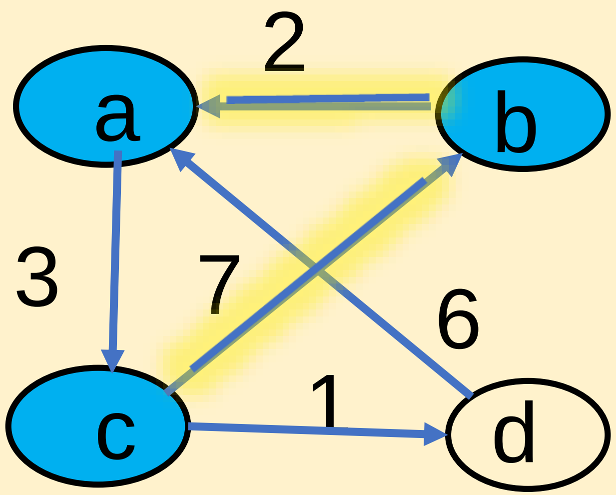


$$D^1 = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

$$D^1 = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \min(7+2, \infty) & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

$c \to b \to a$

Introduction of an intermediate vertex *c* gives two detours



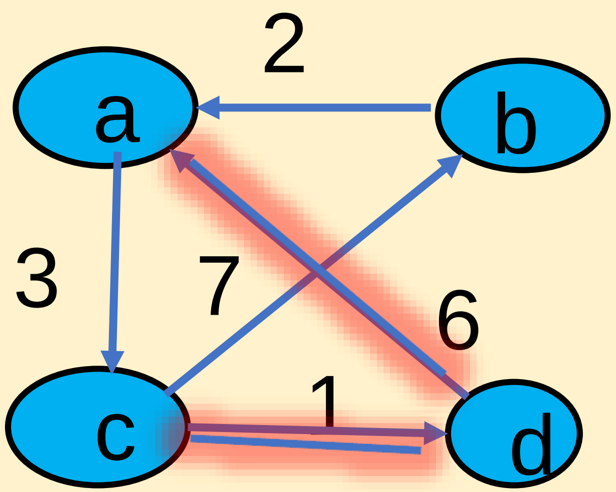
$$D^1 = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

$$D^1 = \begin{bmatrix} 0 & \min(3+7, \infty) & 3 & \min(3+1, \infty) \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} = \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

$a \rightarrow c \rightarrow b$

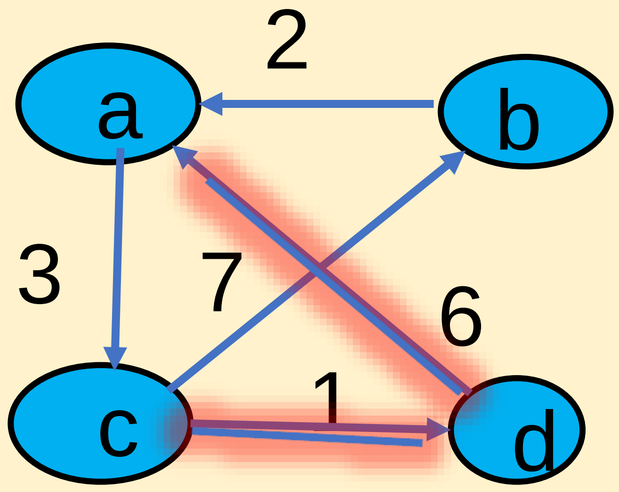
$a \rightarrow c \rightarrow d$

Introduction of an intermediate vertex *d* gives a detour



$$D^1 = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

$$\underset{c \to d \to a}{D^1} = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & \infty \\ \min(1+6, 9) & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & \infty \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$



$$D^1 = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & \infty \\ 7 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix}$$

Introduction of an intermediate vertex ***a and c*** give two detours

$$D^2 = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & \min(2+3+1,\infty) \\ 7 & 7 & 0 & 1 \\ 6 & \min(6+3+7,\infty) & 9 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

$b \to {\color{red}a} \to {\color{red}c} \to d$

$d \to {\color{red}a} \to {\color{red}c} \to b$

More examples

D&C

- ConvexHull
- Matrix Multiplication : Strassen's
- N-bit Multiplication
- •Master's theorem

Greedy Technique

- Prim's MST
- Fractional knapsack
- Single Source Shortest Path

Additional Data structures

- Heapsort
- Proof of BFS
- DFS
- Connected components
- Spanning trees

Dynamic Programming

- Transitive closure
- String Edit
- TSP