# Introduction to Artificial Intelligence

LECTURE 3:

Faster Complete Search

# Overview

- Dynamic Programming
- Minimum Spanning Trees
- Prim’s Algorithm
- Shortest Path Problem
- Djikstra’s Algorithm
- A* Search

# Dynamic Programming

- Synthesises complete solution from partial solutions

  **Synthesisability Condition:** (when) complete solutions *can* be built up from partial solutions.

- Bottom-up:
  - starts from a null solution
  - solves a base case (or cases)
  - composes larger partial solutions, until done

- Often allows for a trade-off: use extra storage for intermediate results; avoid re-computing intermediate results

# Fibonacci Sequence

Fibonacci problem clearly is synthesisable:

$$f(n) = f(n-1) + f(n-2)$$

So, to compute FIB more quickly, we can save intermediate partial solutions rather than re-compute them:

---

(1) INT FIB (INT $n$)

(2) INT $a[0:n]$

(3) $a[0] \leftarrow 0$; $a[1] \leftarrow 1$;

(4) IF $n \leq 1$ RETURN $n$

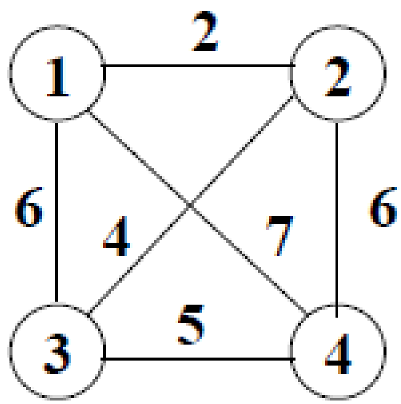
(5) ELSE FOR $i \leftarrow 2$ TO $n$

(6) $\quad a[i] \leftarrow a[i-1] + a[i-2]$;

(7) RETURN $a[n]$;

# TSP - Dynamic Programming

Consider a simple (symmetric) TSP problem:




$$= \mathbf{L} = \begin{pmatrix} \mathbf{0} & & & \\ \mathbf{2} & \mathbf{0} & & \\ \mathbf{6} & \mathbf{4} & \mathbf{0} & \\ \mathbf{7} & \mathbf{6} & \mathbf{5} & \mathbf{0} \end{pmatrix}$$

where $L(i, neighbour(i))$ = (symmetric) edge weight between the two nodes

# TSP - Dynamic Programming

Let $g(i, S)$ =

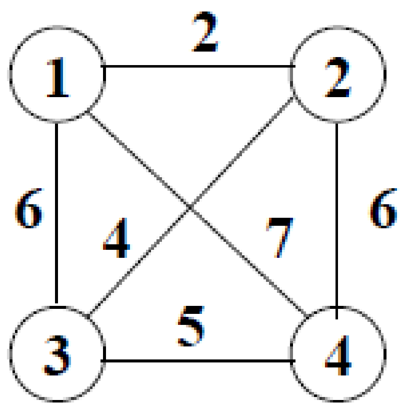
> the length of the shortest path from $i$ to 1 visiting each city in $S$ exactly once (where $S$ is a sub-graph of some graph) and possibly also going through one or more of any cities in the graph which are not in $S$

Goal: Find $g(1, V - \{1\})$ for some graph, $V$

*Synthesisability Condition:*

$$(*) \qquad g(i, S) \quad = \quad \min_{j \in S}\{L(i, j) + g(j, S - \{j\})$$

# TSP - Dynamic Programming




$$= \quad \mathbf{L} \quad = \begin{pmatrix} 0 & & & \\ 2 & 0 & & \\ 6 & 4 & 0 & \\ 7 & 6 & 5 & 0 \end{pmatrix}$$

The circuits and their costs are

$$\begin{aligned} \langle 1234 \rangle &= 18 \\ \langle 1243 \rangle &= 19 \\ \langle 1324 \rangle &= 23 \\ \langle 1342 \rangle &= 19 \\ \langle 1423 \rangle &= 23 \\ \langle 1432 \rangle &= 18 \end{aligned}$$

$$\begin{aligned} g(1, V - \{1\}) &= \min_{j \in V - \{1\}} \{L(1,j) + g(j, V - \{1,j\})\} \\ &= L(1,2) + g(2, V - \{1,2\}) \\ &= L(1,2) + g(2, \{3,4\}) \\ &= 2 + \min(16, 17) \\ &= 18 \end{aligned}$$

# TSP - Dynamic Programming

Idea: solve the base case; figure out how to synthesise sub-answers into larger answers.

Base case: $S = \emptyset$

$$\begin{aligned} g(1, \emptyset) &= 0 \\ g(2, \emptyset) &= 2 \\ g(3, \emptyset) &= 6 \\ g(4, \emptyset) &= 7 \end{aligned}$$

$|S| = 1$

$$\begin{aligned} g(2, \{3\}) &= 10 \\ g(2, \{4\}) &= 13 \\ g(3, \{2\}) &= 6 \\ g(3, \{4\}) &= 12 \\ g(4, \{2\}) &= 8 \\ g(4, \{3\}) &= 11 \end{aligned}$$

And iterate . . .

# TSP - Dynamic Programming Algorithm

---

INT TSP $(L[,], P[,])$

INT $g[,], n \leftarrow dim(L);$

FOR $i = 1$ TO $n$ $g[i, \emptyset] \leftarrow L[i, 1];$

FOR $k = 1$ TO $n - 2$

    [FOR ALL $S \subseteq V - \{1\}$ S.T. $|S| = k$

        [FOR $i \notin S \cup \{1\}$

            $[g[i, S] \leftarrow \min_{j \in S}(L[i, j] + g[j, S - \{j\}]);$

            $P[i, S] \leftarrow$ THAT $j$;]]]

$g[1, V - \{1\}] \leftarrow \min_{j \in V - \{1\}}(L[1, j] + g[j, V - \{1, j\}]);$

$P[1, V - \{1\}] \leftarrow$ THAT $j$;

RETURN $g[1, V - \{1\}];$

---

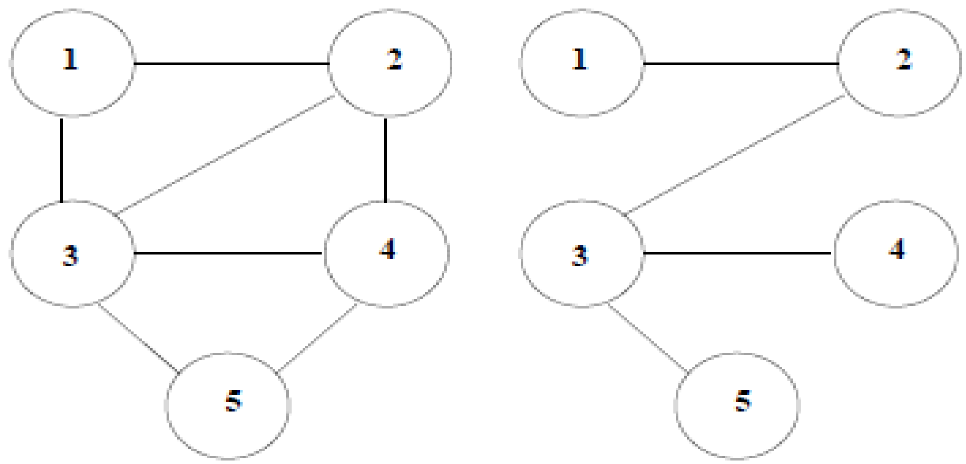
Return value is the min length; returned $P$ reports the min path.

# TSP : DP VS Complete Search

| Cities | DP $[2^{n-3}]$ | Brute Force $[\ (n-1)!/2]$ |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 2 | 3 |
| 3 | 4 | 12 |
| 4 | 8 | 60 |
| 5 | 16 | 360 |
| 6 | 32 | 2520 |
| 7 | 64 | 20160 |
| 8 | 128 | 181440 |
| 9 | 256 | 1814400 |
| 10 | 512 | 19958400 |
| 11 | 1024 | 239500800 |
| 12 | 2048 | 3113510400 |
| 13 | 4096 | 43589145600 |
| 14 | 8192 | 653837184000 |
| 15 | 16384 | 10461394944000 |
| 16 | 32768 | 177843714048000 |
| 17 | 65536 | 3201186852864000 |
| 18 | 131072 | 60822550204416000 |

# Spanning Trees

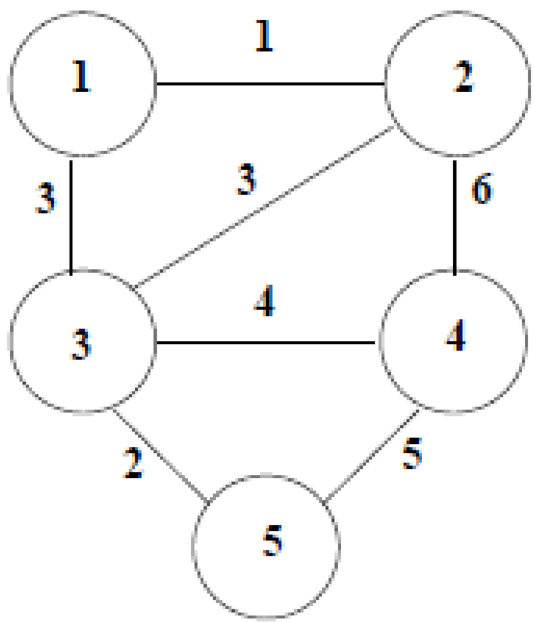
- Definition 1: Connected Graph
  - A **connected** graph is one in which there is a path from each node to every other node.




- Definition 2: Tree
  - A **tree** is an acyclic, connected graph
- Definition 3: Spanning Tree
  - A **spanning tree** is a tree that contains all the nodes of the original graph.

# Minimum Spanning Tree

- The minimum spanning tree problem is to find a spanning tree which minimizes the sum of weights on a spanning tree.



# Minimum Spanning Tree Algorithm

(1) $V' := random(V)$; [CHOOSE THE ROOT]

(2) $E' := \emptyset$

(3) WHILE $V' \neq V$ DO

(4) $\quad [E' := E' \cup E(nearest(V', V \backslash V'))$;

(5) $\quad V' := V' \cup nearest(V', V \backslash V')]$

- *nearest* return the nearest node not in V'
- E(nearest) returns the edge to that node

# Prim's Algorithm

- Outer loop executes n-1 times
- Let us examine *nearest*

---

(1) $min := \infty; next := null;$

(2) FOR $v \in V - V'$

(3) FOR $w \in V'$

(4) [IF $d(v, w) < min$

(5) THEN [$next := v; min := d(v, w)$]];

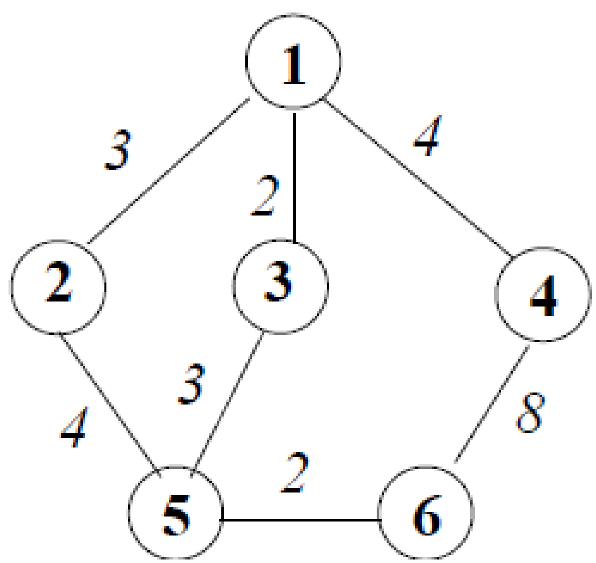
(6) IF $next \neq null$ RETURN $next$ ELSE FAIL;

---

- Inner loop (e.g., (4) and (5) above) is executed $(n - k)k$ times, where $k = |V'|$.
- This is maximized when $k = n/2$ (differentiate $(n - k)k$ wrt k and set to zero), when it executes $n^2/4$ times.
- So, this algorithm is $O(n^3)$

# Shortest Path Problem

- Problem formulation
  - For any two nodes v,w ∈ V find the shortest path between them.
- Djikstra's Algorithm (vaguely analogous to Prim's)
  - Expand the partial path with the lowest cost until the complete solution has been built.
- Example: What is the shortest path from node 1 to 6?



# Djikstra's Algorithm

(1) $open \leftarrow initial; closed \leftarrow \langle\rangle;$

(2) LOOP: IF $open = \langle\rangle$ FAIL;

(3) $v \leftarrow pop(open);$

(4) $push(v, closed);$

(5) IF $goal(v)$ THEN RETURN $path(initial, v)$

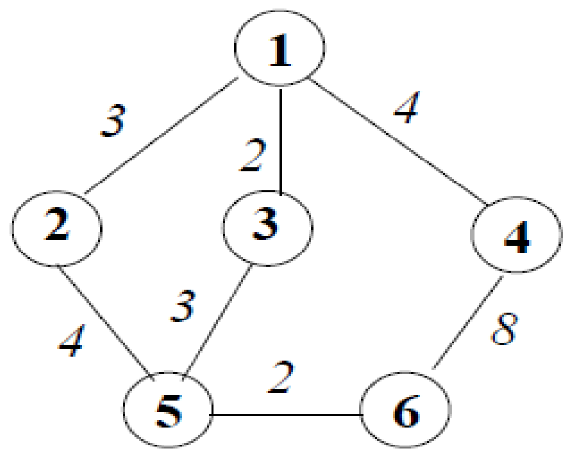
(6) ELSE $open \leftarrow sort(children(v), open);$

(7) GO LOOP;

- *initial* contains the starting node
- *goal* is a test for the target node
- The sort operation sorts nodes by retrospective pathcost – lowest cost of the partial path to the node being added to *open*.

# Djikstra's Algorithm

Example:




At step 2 the open and closed lists will be:

| | |
|---|---|
| $\langle 1 \rangle$ | $\langle \rangle$ |
| $\langle 324 \rangle$ | $\langle 1 \rangle$ |
| $\langle 245 \rangle$ | $\langle 13 \rangle$ |
| $\langle 45 \rangle$ | $\langle 132 \rangle$ |
| $\langle 56 \rangle$ | $\langle 1324 \rangle$ |
| $\langle 6 \rangle$ | $\langle 13245 \rangle$ |
| $\langle \rangle$ | $\langle 132456 \rangle$ |

# Djikstra's Algorithm

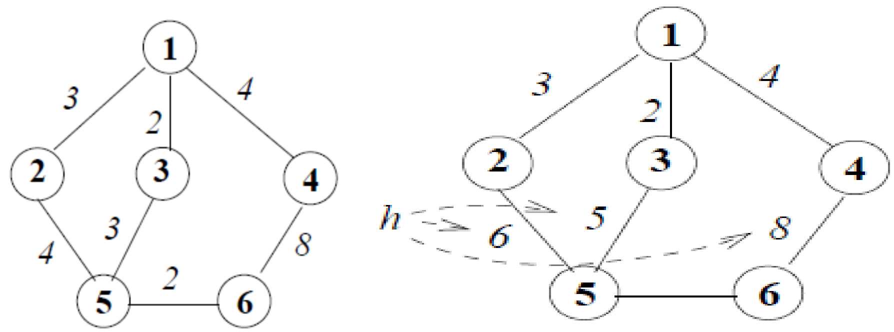
- Dynamic programming synthesizing approach (complete search)
  - Each partial path is guaranteed to have minimum cost when added; therefore, when the final path is added it will have minimum cost.
  - $O(n^2)$
- Greediness?

# A* Search Algorithm

- Generalization of Dijkstra's algorithm by considering heuristic evaluation
  - Sorting based on
    - $c(x)$ = retrospective cost from $i$ to $x$
    - $h(x)$ = prospective heuristic cost from $x$ to the nearest goal
    - $eval(x) = c(x) + h(x)$ = $A^*$ eval of the path
    - $best(X) = \min_{x \in X} \{eval(x)\}$
- Note: this is "heuristic" only in the sense of using a heuristic evaluation function. Not in the original sense of a hit-or-miss heuristic: it is *complete*!

# A* Search Algorithm

E.g., suppose our heuristic $h$ is perfect. Then




At step 2 the open and closed lists will be:

| | |
|---|---|
| $\langle 1 \rangle$ | $\langle \rangle$ |
| $\langle 324 \rangle$ | $\langle 1 \rangle$ |
| $\langle 524 \rangle$ | $\langle 13 \rangle$ |
| $\langle 624 \rangle$ | $\langle 135 \rangle$ |
| $\langle 24 \rangle$ | $\langle 1356 \rangle$ |

Qualitatively (check visually on larger graphs):

- Dijkstra's algorithm spreads search out in all directions
- $A^*$ (with a good $h$) focuses search along a promising set of paths (in a "finger")

# A* Search Algorithm

- Quality of the algorithm depends upon the heuristic evaluation function h.
- The main issue is to generate an *admissible* heuristic.
  - $h$ is **admissible** iff $\forall S \; h(S) \leq dist(S, G)$
- Theorem: A* is complete iff. its heuristic is admissible
- Given admissibility, efficiency of the algorithm improves as h(x) converges on the true cost to goal.

# Both Djikstra's and A* Algorithms are not Greedy

- Informal 'Proof'
  – Greedy algorithms make the best looking single step from wherever they are now.
  – In A$\ast$ the best looking single step is given by h(x); however, A$\ast$ takes that only if h(x) + c(x) happens to be best.
  – Dijkstra's takes the best looking next step only if the cost to it plus total retrospective cost happens to be best.
  – Incidentally, Dijkstra could only be greedy if A$\ast$ is greedy, since it *is* A$\ast$ with h(x) = 0 everywhere.
- A$\ast$ is systematic and complete (but, given a good h, not exhaustive).

# 3 Kinds of Search

1. *Complete (Optimal) Search:* Guaranteed to find the goal (optimum), if there is one. E.g.,
   - Exhaustive search
2. *Heuristic Search:* Not necessarily guaranteed to find the (optimal) answer. E.g.,
   - Greedy search
3. *Stochastic Search:* probabilistic search through the state space. Not guaranteed, hence a (sub)kind of heuristic search. E.g.,
   - Genetic algorithm, simulated annealing