- **Greedy algorithms**
  - **General method**
    - ✓ **Examples**
    - ✓ **Control abstraction**
  - **Fractional Knapsack Problem**

General method

- A greedy algorithm refers to any algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage.
- It is a mathematical process which looks for simple, easy-to-implement solutions in stages.
- The decision is made to provide the most obvious benefit.
- In many problems, a greedy strategy does not usually produce an optimal solution
- But may still yield locally optimal solutions that approximate a globally optimal solution quickly.

Subset paradigm

- Given n inputs choose a subset that satisfies some constraints.
- A subset that satisfies constraints is called a feasible solution.
- A feasible solution that maximises or minimises a given (objective) function is said to be optimal.
- Often it is easy to find a feasible solution but difficult to find the optimal solution.
- The greedy method suggests that one can devise an algorithm that works in stage. At each stage a decision is made whether a particular is in the optimal solution.
- This is called subset paradigm.

Examples --- Change problem

- A child buys candy valued at less than £1 and gives a £1 bill to the cashier.
- The cashier wishes to return change using the fewest number of coins.
- The cashier constructs the change in stages. At each stage increase the total amount of change as much as possible.
- The added coin should not cause the total amount of change given so far to exceed the final desired amount (feasibility).
- Suppose that 67 pence is due to the child. The first coin selected are 50 pence.
- The second coin cannot be a 50p or 20p as not feasible.
- The second is 10 pence, then a 5 pence, and finally two pence are added to the change.

Examples --- Knapsack problem

- Your train breaks down in a desert and you decide to walk to nearest town.
- You have a rucksack but which objects should you take with you ?
- Feasible: Any set of objects is a feasible solution provided that they are not too heavy, fit in the rucksack and will help you survive (these are constraints).
- An optimal solution is the one that maximises or minimises something
    - One that minimises the weight carried
    - One that fills the rucksack completely (maximise)
    - One that ensures the most water is taken etc

Other examples:

- You want to work out the best way to route a phone message through a mobile phone network.
- A number of users want to run programmes on a computer. How do you schedule them so they are executed as quickly as possible.
- A factory use a production line to make several products. How should you schedule the production runs to make the most profit.
- You run a haulage company and want to workout how to deliver all your products to a set of outlets with the least cost and time.
- You run an airline and want to work out how best to turn the plane around on landing and get it flying again.

Control abstraction for Greedy Algorithm

```
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
}
```

- The function *Greedy* describes the essential way that a greedy algorithm will look, once a particular problem is chosen functions *Select, Feasible, and Union* are properly implemented.

- The function *Select* selects an input from *A* whose value is assign to *x*.
- *Feasible* is a Boolean-valued function that determines if *x* can be included into the solution vector.
- The function *Union* combines *x* with the solution, and update the objective function.

```
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
}
```

Kruskal MST

Fractional Knapsack

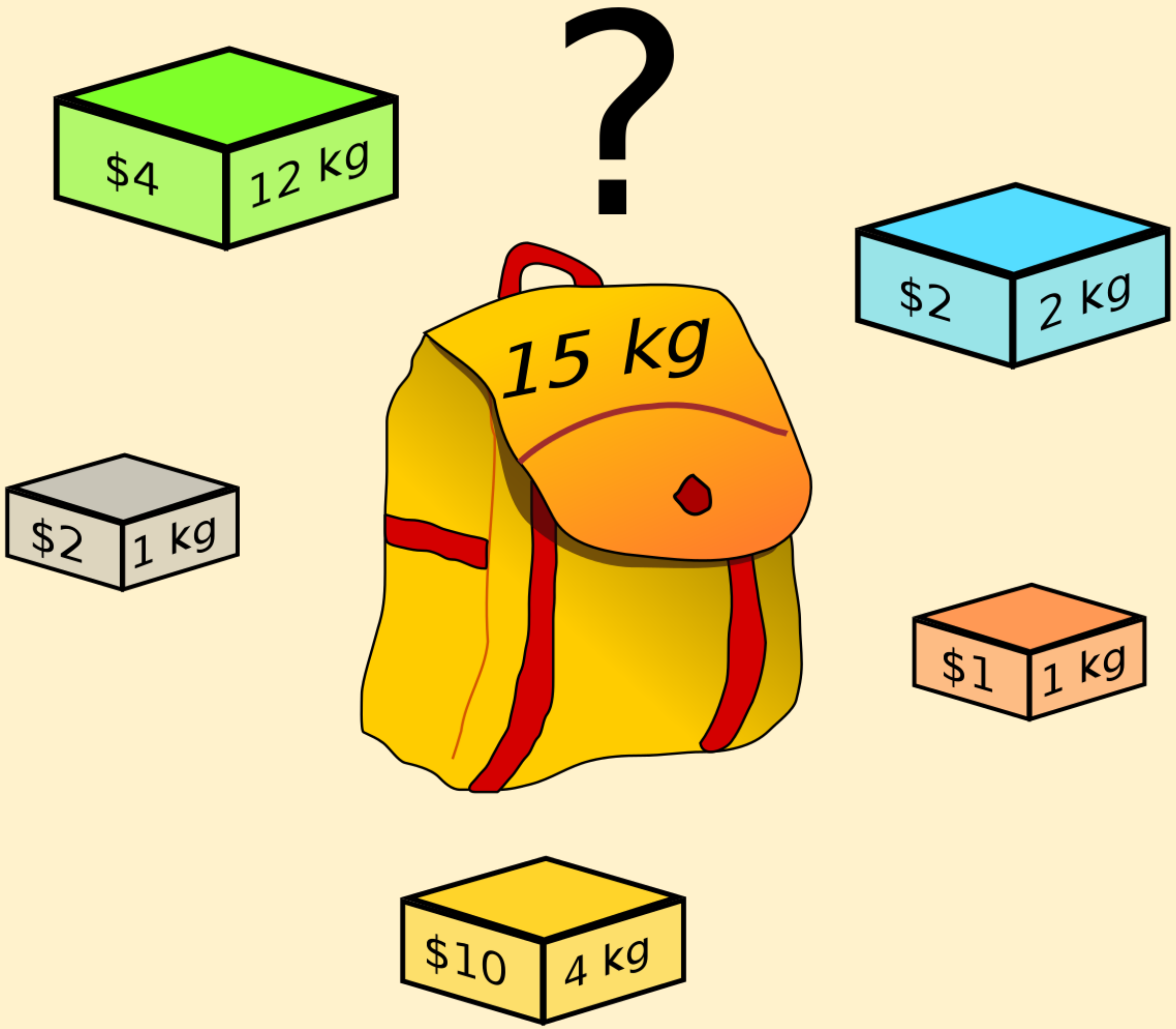
Greedy Algorithms

Prim MST

Dijkstra shortest path

Fractional Knapsack Problem

- Given n objects and a knapsack (or rucksack) with a capacity (weight) M
- Each object i has weight w_i, and profit p_i
- For each object i, suppose a fraction $x_i, 0 < x_i \leq 1$ (i.e. 1 is the maximum amount) can be placed in the knapsack, then the profit earned is $p_i x_i$



- Objective is to maximize profit subject to capacity constraint. i.e. Maximize

$$\sum_{i=1}^{n} p_i x_i \qquad (1)$$

- Subject to

$$\sum_{i=1}^{n} w_i x_i \le M \qquad (2)$$

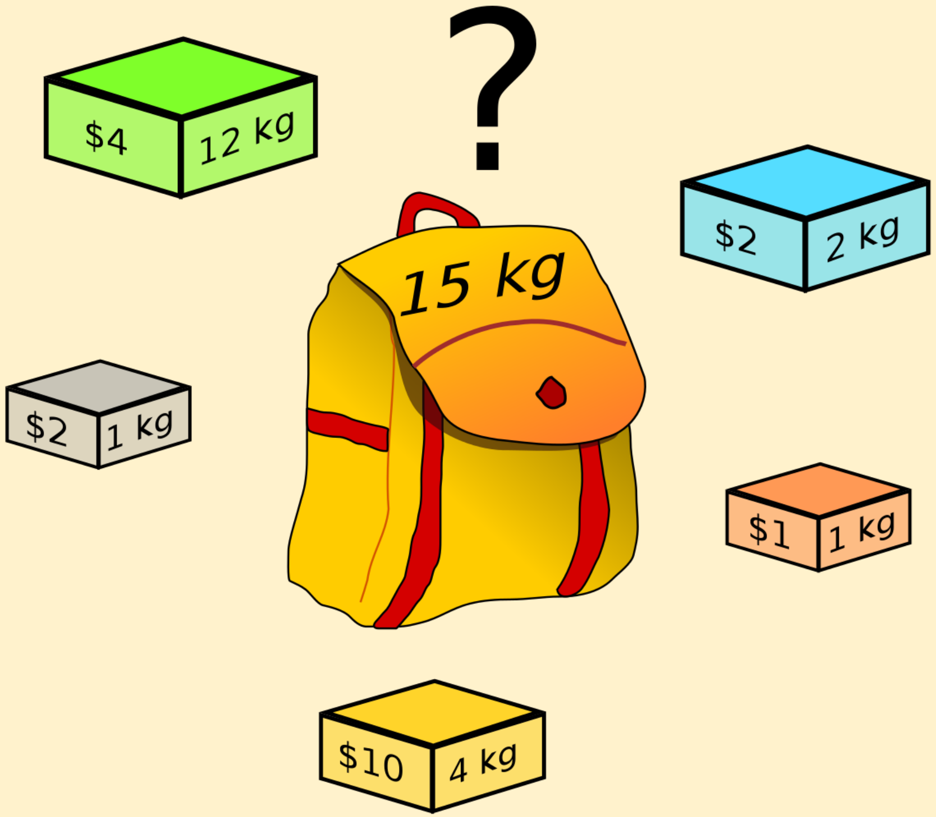
$$0 \le x_i \le 1$$

$$p_i > 0 \qquad (3)$$

$$w_i > 0$$

- A feasible solution is any subset $\{x_1 \cdots\cdots x_n \}$ satisfying (2) and (3).
- An optimal solution is a feasible solution that maximize (1)

- Knapsack problems appear in real-world decision-making processes in a wide variety of fields, such as
  - finding the least wasteful way to cut raw materials
  - selection of investments and portfolios,
  - resources allocation, etc.



Example Let $n = 3; M = 20$

$(p_1 , p_2 , p_3) = (25,24,15)$

$(w_1 , w_2 , w_3) = (18,15,10)$

Feasible Solutions

	(x_1 , x_2 , x_3)	$\sum_{i=1}^{n} w_i x_i$	$\sum_{i=1}^{n} p_i x_i$

Strategy 1 : maximise objective function

$n = 3; M = 20$
$(p_1 , p_2 , p_3) = (25,24,15)$
$(w_1 , w_2 , w_3) = (18,15,10)$

$$\sum_{i=1}^{n} p_i x_i = 25 \times 1 + 24 \times \frac{2}{15} + 0 = 28.5$$

$$\sum_{i=1}^{n} w_i x_i = 18 \times 1 + 15 \times \frac{2}{15} + 0 = 20$$

- Capacity was quickly exhausted which constrained the profit attained

- Put the object with the greatest profit in the knapsack.
- Then use a fraction of the last object to fill the knapsack to capacity.
- Strategy does not yield an optimal solution.

Strategy 2 : maximise capacity

$n = 3; M = 20$

$(p_1 , p_2 , p_3) = (25,24,15)$

$(w_1 , w_2 , w_3) = (18,15,10)$

$$\sum_{i=1}^{n} p_i x_i = 0 + 24 \times \frac{2}{3} + 15 \times 1 = 31$$

$$\sum_{i=1}^{n} w_i x_i = 0 + 15 \times \frac{2}{3} + 10 \times 1 = 20$$

- Rate of increase of profit was not high enough

- Choose objects according to least weight.
- The idea is that we will get more objects into the knapsack and potentially more profit.
- Solution is still not optimal.

Strategy 3 : balancing profit and capacity

$n = 3; M = 20$

$(p_1 , p_2 , p_3) = (25,24,15)$

$(w_1 , w_2 , w_3) = (18,15,10)$

$$\sum_{i=1}^{n} p_i x_i = 0 + 24 \times 1 + 15 \times \frac{1}{2} = 31.5$$

$$\sum_{i=1}^{n} w_i x_i = 0 + 15 \times 1 + 10 \times \frac{1}{2} = 20$$

- Achieves a balance between rate at which profit increases with the rate at which the capacity is used.

- Find the object to include by maximum profit per unit of capacity. i.e compute $\frac{P_i}{w_i}$.
- Then choose objects starting with the largest and working to smallest ratio.
- Solution is optimal!

- *Input: the objects are in increasing order so that* $\frac{P[i]}{w[i]} > \frac{P[i+1]}{w[i+1]}$
- *Output: optimal solution vector* x
- *causes over flow*
- *greedy choice*
- *Choose a fraction*

```
Algorithm Knapsack(P, W, x:arrayvals; M; n:int)

    for(i= 1;i <=n;i+ +)   x[i] = 0;
    capacity=M;

    for(i= 1;i <=n;i+ +){
      if    W[i]> capacity then exit()
      else
          x[i] = 1;
          capacity=capacity−W[i];
    }
    if i <=n then x[i] =capacity/W[i];
}
```

- **Greedy algorithms**
  - **Single source shortest path problem**
  - **Dijkstra's shortest path algorithm**

Control abstraction for Greedy Algorithm

```
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
}
```

- The function *Greedy* describes the essential way that a greedy algorithm will look, once a particular problem is chosen functions *Select, Feasible, and Union* are properly implemented.

```
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
}
```

- The function *Select* selects an input from *A* whose value is assign to *x*.
- *Feasible* is a Boolean-valued function that determines if *x* can be included into the solution vector.
- The function *Union* combines *x* with the solution, and update the objective function.

Kruskal MST

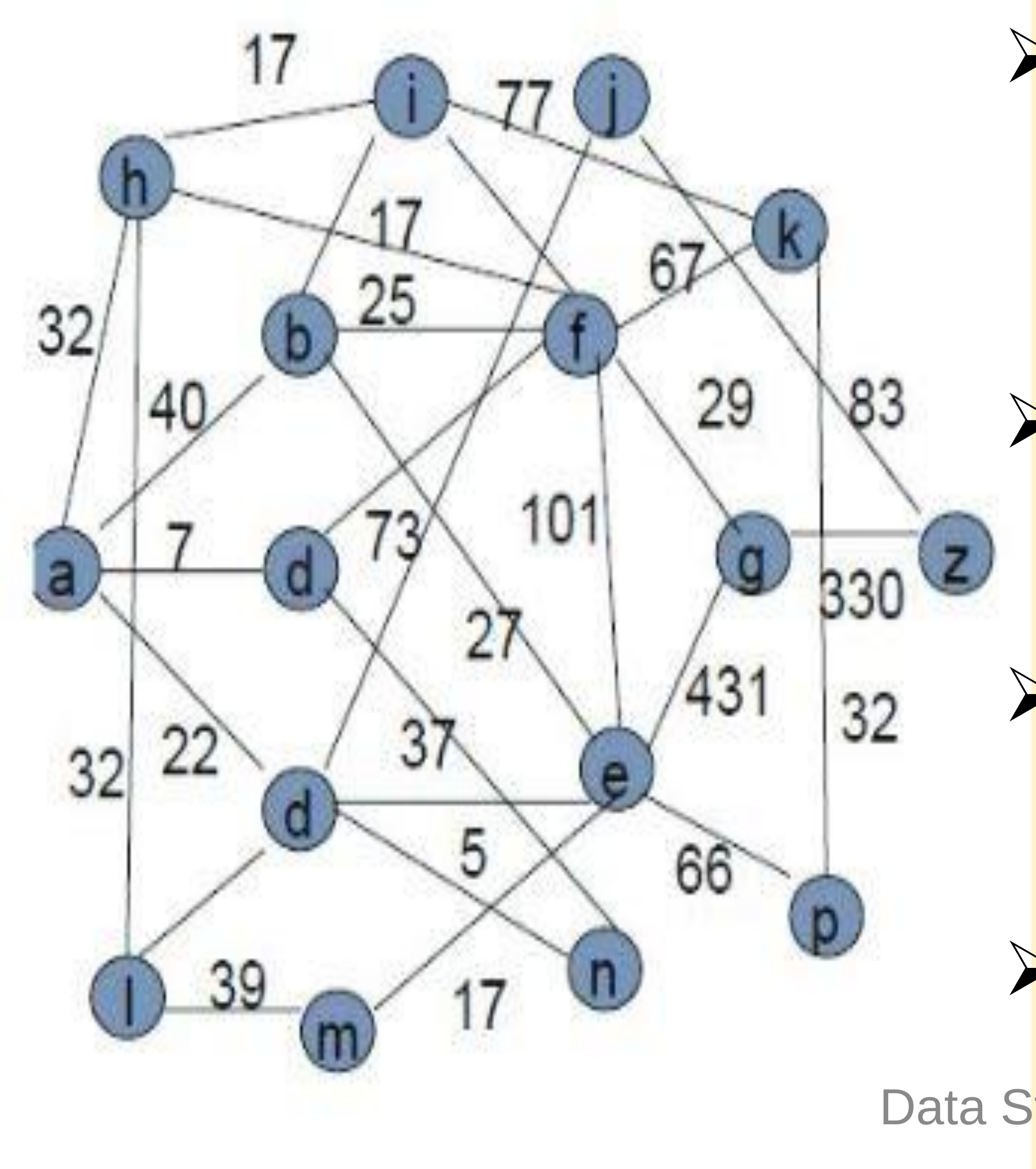
Fractional Knapsack

Greedy Algorithms

Prim
MST

Dijkstra shortest path

Single source shortest path problem



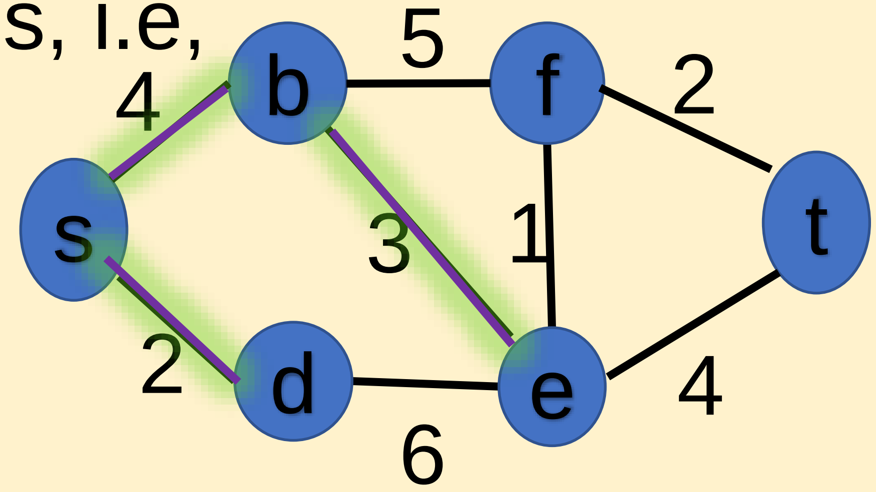
- In graph theory, the problem of finding the shortest path between two nodes on a graph is called shortest path problem.
- The graph type is weighted graph, the number attached to each edge is a weight.
- Single source shortest path solves the shortest path from a given vertex
- Given a weighted graph, find the shortest path from h to z ?

- ➢ Graphs naturally represent networks, e.g. Road/Rail/Air networks, oil pipelines, electricity grids.
- ➢ Modern day applications (AA route planner, Sat Nav , Mobile Maps)
- ➢ Natural questions are
  - Find a route or path from city A to city B
  - Different paths with lowest cost (e. g least fuel, least distance, least travel time)

➢ Other applications include plant and facility layout, robotics, transportation, and VLSI design.

- <u>Definition:</u> 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,

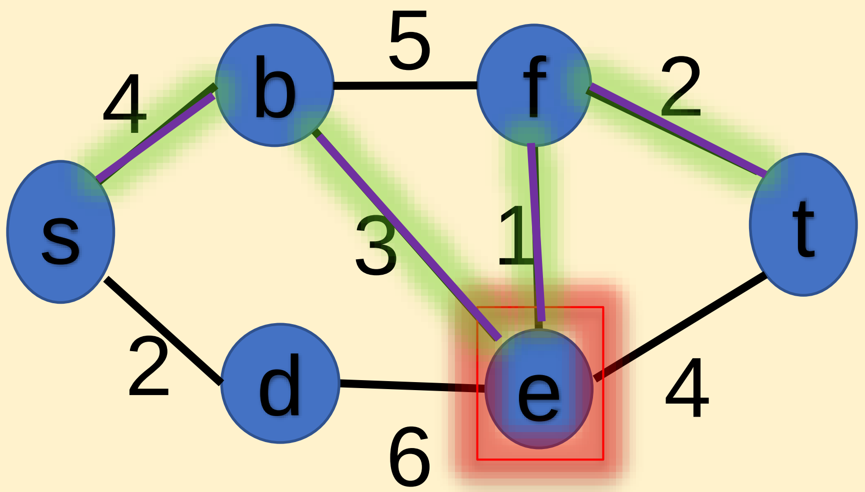
$$w(P) = \sum_{i=1}^{k} e_i$$



- 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

- This is the most important algorithms for solving the single-source shortest path problem with non-negative edge weight.

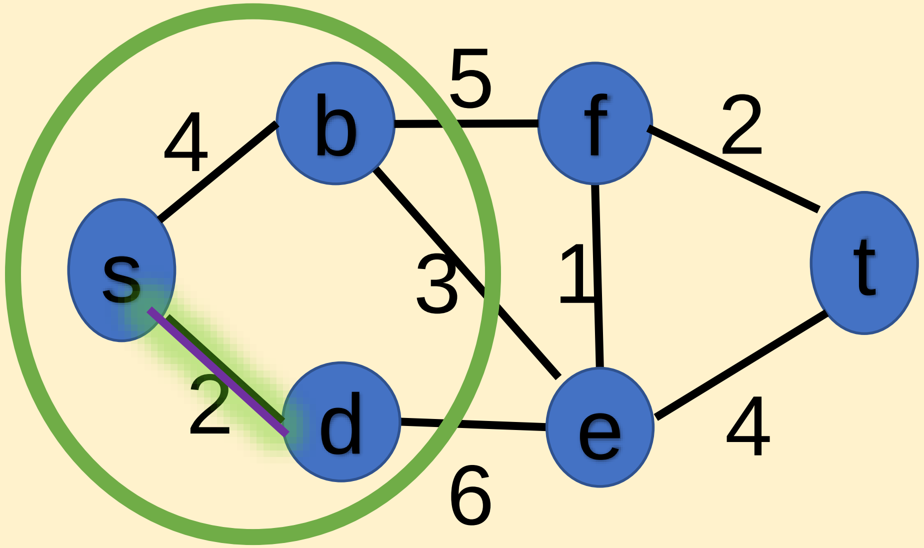


- Dijkstra's algorithm is based on the property that if a shortest path from s to t goes through vertex e
  - then the sub-path from s to e is a shortest path from s to e.
  - and the sub-path from e to t is a shortest path from e to t

Algorithm outline

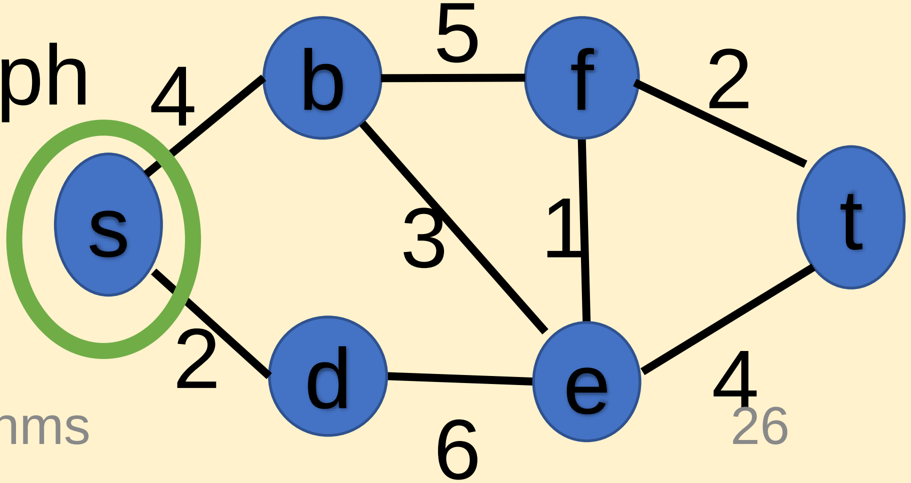
- Maintain a set S of vertices u for which a shortest path distance $Dist(u)$ has been determined from a starting vertex s.

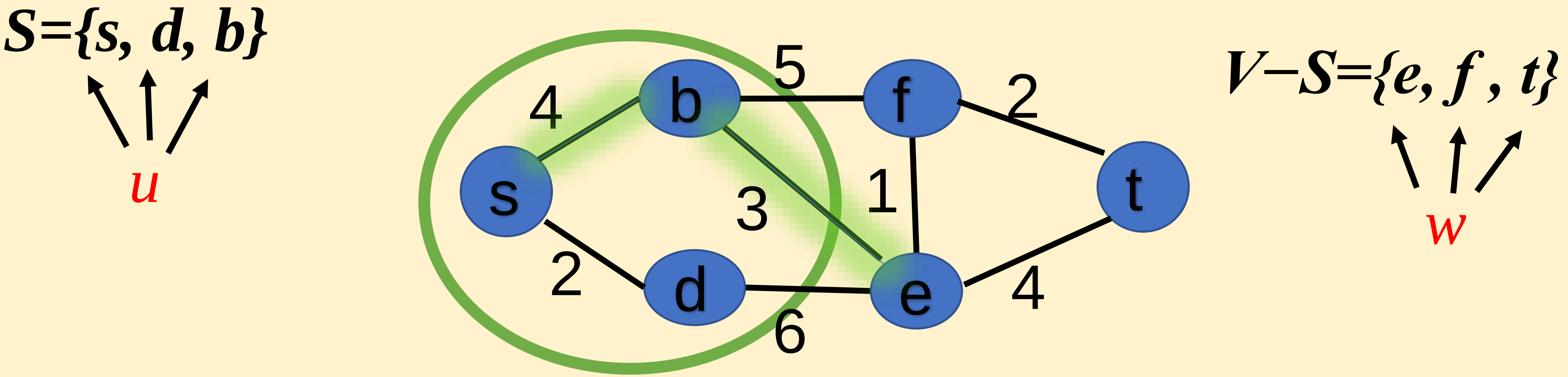
$S=\{s, d, b\}$



$V-S=\{e, f, t\}$

- This is the "explored" part of the graph
- Initially $S= \{s\}$ and $Dist(s) = 0$





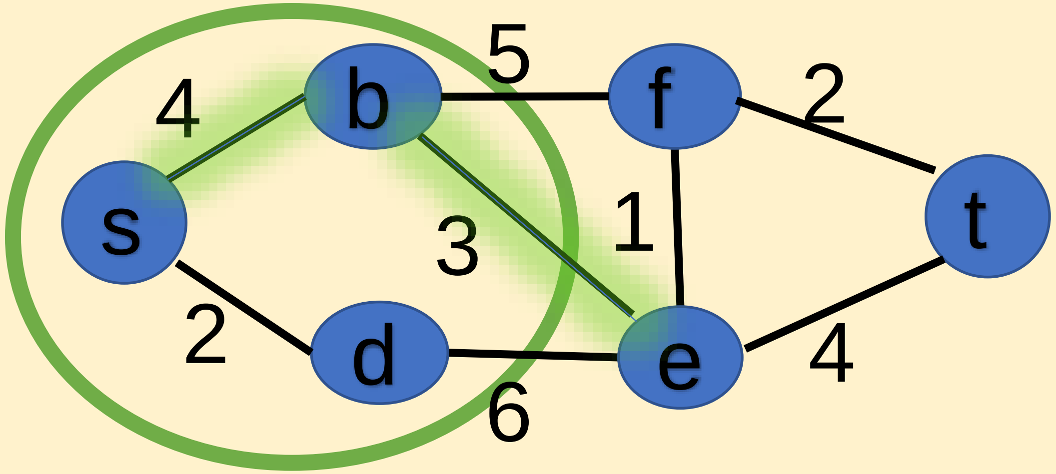
- For each vertex w in $\{V\text{-}S\}$, determine the shortest path starting from s travelling along path through the explored part S to some vertex u followed by an edge (u, w).
- The destination w is such that

$$Dist(w) = \min_{u \in S,\ w \in V-S} (Dist(u) + \mathrm{cost}(u, w))$$

- i.e. Choose the node w for which the quantity $Dist(u) + \mathrm{cost}(u, w)$ is minimized

$S=\{s, d, b\}$

u



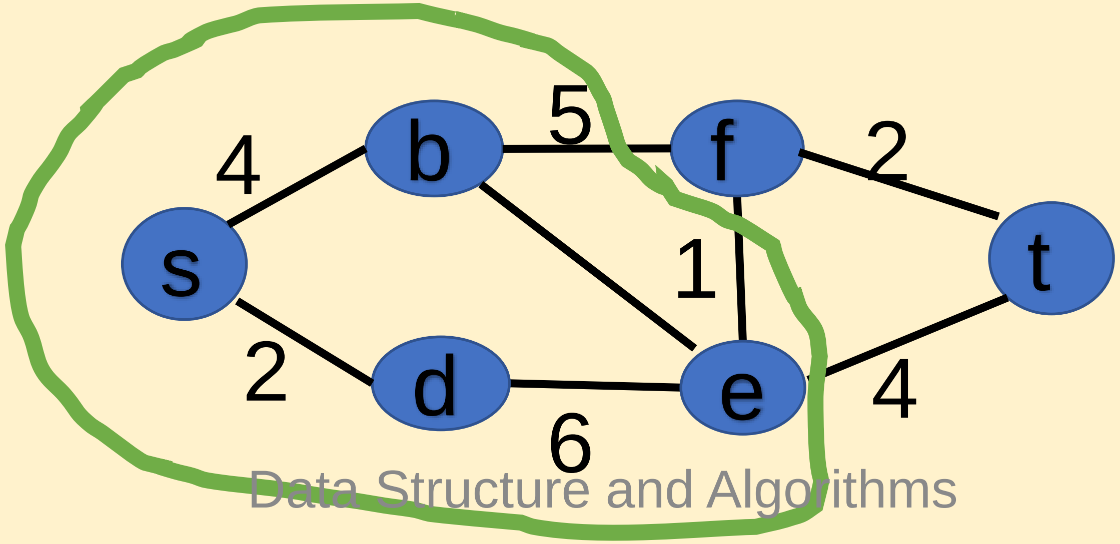
$V-S=\{e, f, t\}$

w

- Add w to set S and repeat the above procedure until the destination is reached.

$S=\{s, d, b, e\}$

u



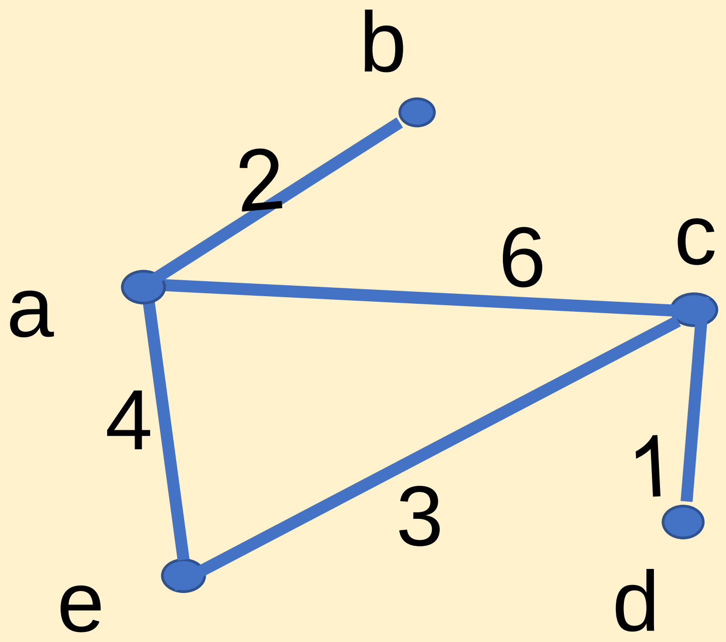
$V-S=\{f, t\}$

w

- Need also an array a shortest path distance *Dist(u)* has been determined from a starting vertex.
- The algorithm is greedy, with the set *S* increases one element at a time.
- Need an indicator array to record which node is in *S*
- The adjacency matrix of weighted graph is used in the algorithm to represent the graph as input for problem solving.

➢ *A Recap:* Adjacency matrix for weighted graph

$$A[i,j] = \begin{cases} c & \text{if edge } <i, j> \text{ is in } E(G) \\ \infty & \text{otherwise} \end{cases}$$

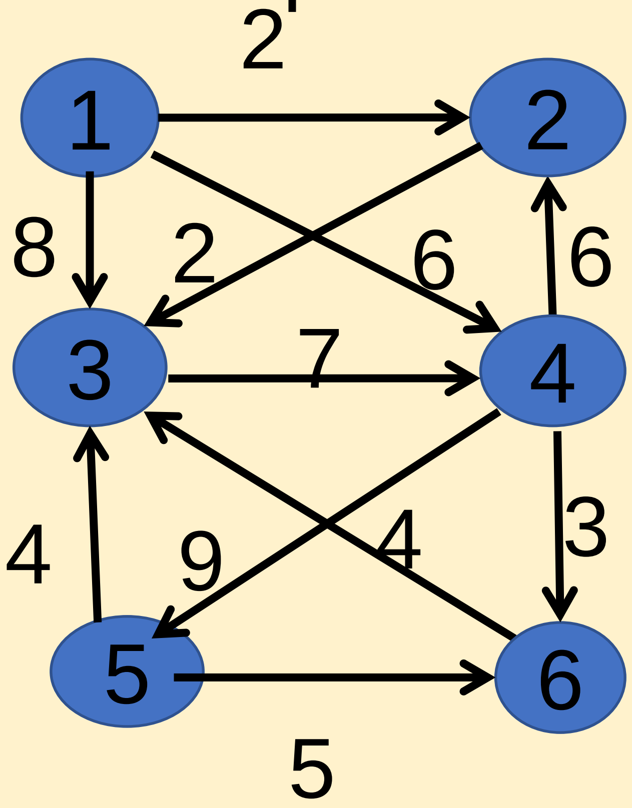


$$\begin{bmatrix} 0 & 2 & 6 & \infty & 4 \\ 2 & 0 & \infty & \infty & \infty \\ 6 & \infty & 0 & 1 & 3 \\ \infty & \infty & 1 & 0 & \infty \\ 4 & \infty & 3 & \infty & 0 \end{bmatrix}$$

```
Algorithm ShortestPaths(v:int; Cost:Matrix; Dist: Array[n]; n:int) {
    for(i= 1; i <=n; i++)
     S[i] = 0; Dist[i] =Cost[v, i];
     S[v] = 1; Dist[v] = 0;
      for(j= 2;j <=n−1;j+ +) {
     choose u such that Dist[u] =min(Dist[w])
      and S[w] = 0;
      S[u] = 1;
            for all w with S[w] = 0 {
           Dist[w] =min(Dist[w], Dist[u] +Cost[u, w]) }
   }
}
```

- *Dist updates the shortest path lengths to each vertex from v.*
- *initially no vertices are in set S and cost of shortest path is for the weight of edge (v, i)*
- *start with vertex v put it in S*
- *determine (n-1) paths from v*
- *add it to S*
- *update path lengths for vertices not in S*

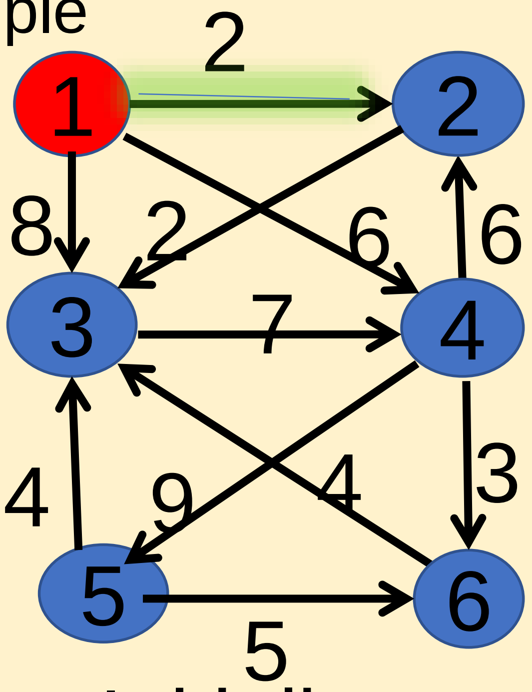
Example



$$Cost = \begin{bmatrix} 0 & 2 & 8 & 6 & \infty & \infty \\ \infty & 0 & 2 & \infty & \infty & \infty \\ \infty & \infty & 0 & 7 & \infty & \infty \\ \infty & 6 & \infty & 0 & 9 & 3 \\ \infty & \infty & 4 & \infty & 0 & 5 \\ \infty & \infty & 4 & \infty & \infty & 0 \end{bmatrix}$$

- Paths from vertex 1 to vertex 6
  - (1,2)(2,3)(3,4)(4,6) 2 + 2 + 7 + 3 = 14
  - (1,3)(3,4)(4,6) 8 + 7 + 3 = 18
  - (1,4)(4,6) 6 + 3 = 9

Example



$$Cost = \begin{bmatrix} 0 & 2 & 8 & 6 & \infty & \infty \end{bmatrix}$$

Initialise $v = 1, S[1] = 1, Dist[1] = 0$

$S[2] = 0, Dist[2] = 2$

$S[3] = 0; Dist[3] = 8$

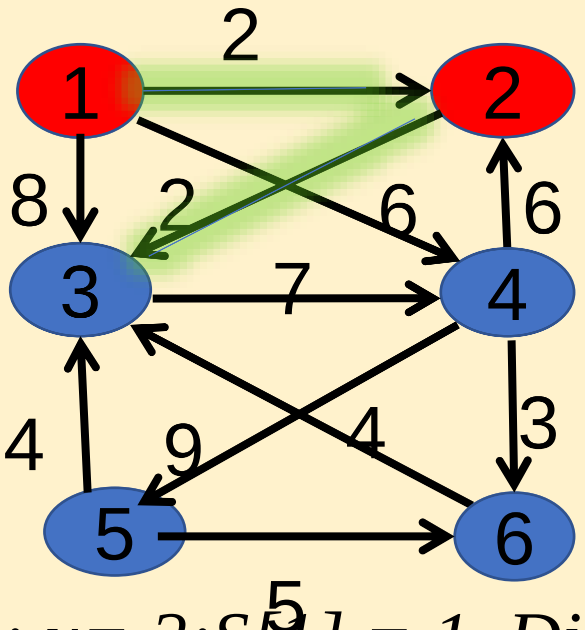
$S[4] = 0; Dist[4] = 6$

$S[5] = 0; Dist[5] = \infty$

$S[6] = 0; Dist[6] = \infty$

Example

$Dist = [0, 2, 8, 6, \infty, \infty]$



$$Cost = \begin{bmatrix} 0 & 2 & 8 & 6 & \infty & \infty \\ \infty & 0 & 2 & \infty & \infty & \infty \end{bmatrix}$$

$j = 2;\ u = 2;\ S[1] = 1,\ Dist[1] = 0$

$S[2] = 1,\ Dist[2] = 2$

$S[3] = 0,\ Dist[3] = \min(8, 2 + 2) = 4$

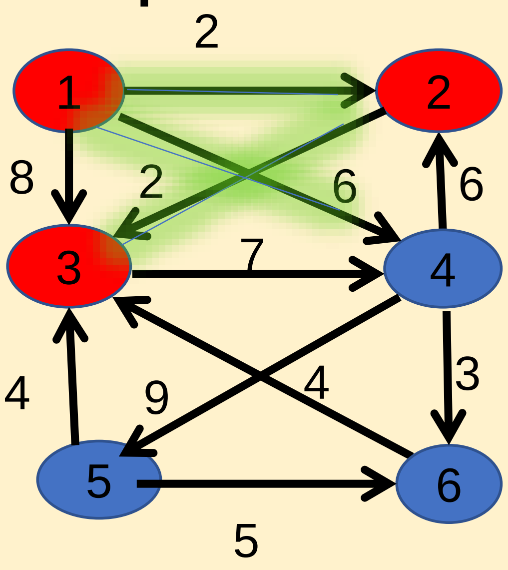
$S[4] = 0,\ Dist[4] = \min(6, 2 + \infty) = 6$

$S[5] = 0,\ Dist[5] = \min(\infty, 2 + \infty) = \infty$

$S[6] = 0,\ Dist[6] = \min(\infty, 2 + \infty) = \infty$

$$Dist = [0, 2, 4, 6, \infty, \infty]$$

Example



$$Cost = \begin{bmatrix} 0 & 2 & 8 & 6 & \infty & \infty \\ \infty & 0 & 2 & \infty & \infty & \infty \\ \infty & \infty & 0 & 7 & \infty & \infty \end{bmatrix}$$

$j = 3; u = 3; S[1] = 1, Dist[1] = 0$

$S[2] = 1, Dist[2] = 2$

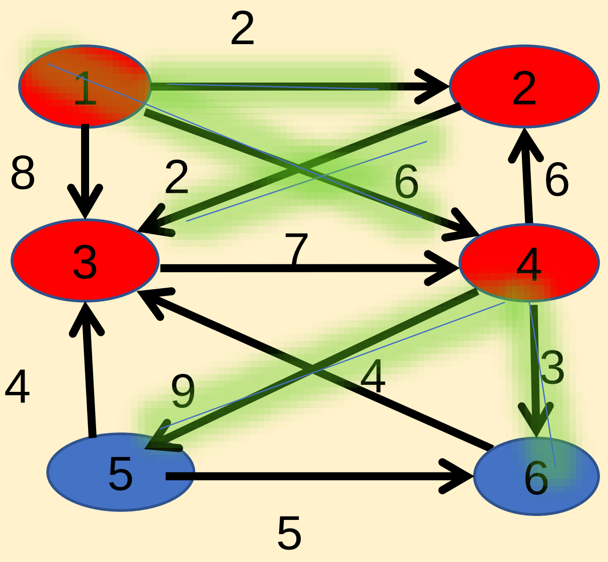
$S[3] = 1, Dist[3] = 4$

$S[4] = 0, Dist[4] = min(6, 4 + 7) = 6;$

$S[5] = 0, Dist[5] = min(\infty, 4 + \infty) = \infty$

$S[6] = 0, Dist[6] = min(\infty, 4 + \infty) = \infty$

Example



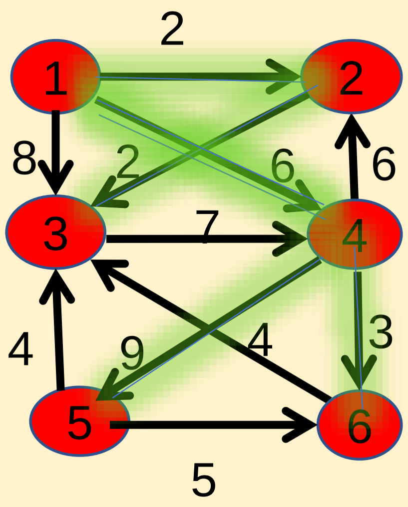
$$Dist = [0, 2, 4, 6, \infty, \infty]$$

$$Cost = \begin{bmatrix} 0 & 2 & 8 & 6 & \infty & \infty \\ \infty & 0 & 2 & \infty & \infty & \infty \\ \infty & \infty & 0 & 7 & \infty & \infty \\ \infty & 6 & \infty & 0 & 9 & 3 \end{bmatrix}$$

$$j = 4, u = 4, \quad S[1] = 1, \; Dist[1] = 0$$
$$S[2] = 1, \; Dist[2] = 2$$
$$S[3] = 1, \; Dist[3] = 4$$
$$S[4] = 1, \; Dist[4] = 6$$
$$S[5] = 0, \; Dist[5] = \min(\infty; 6 + 9) = 15$$
$$S[6] = 0, \; Dist[6] = \min(\infty; 6 + 3) = 9$$

Example

$$Dist = [0, 2, 4, 6, 15, 9]$$



$$Cost = \begin{bmatrix} 0 & 2 & 8 & 6 & \infty & \infty \\ \infty & 0 & 2 & \infty & \infty & \infty \\ \infty & \infty & 0 & 7 & \infty & \infty \\ \infty & 6 & \infty & 0 & 9 & 3 \\ \infty & \infty & 4 & \infty & 0 & 5 \\ \infty & \infty & 4 & \infty & \infty & 0 \end{bmatrix}$$

$j = 5, u = 6, S[1] = 1, Dist[1] = 0$

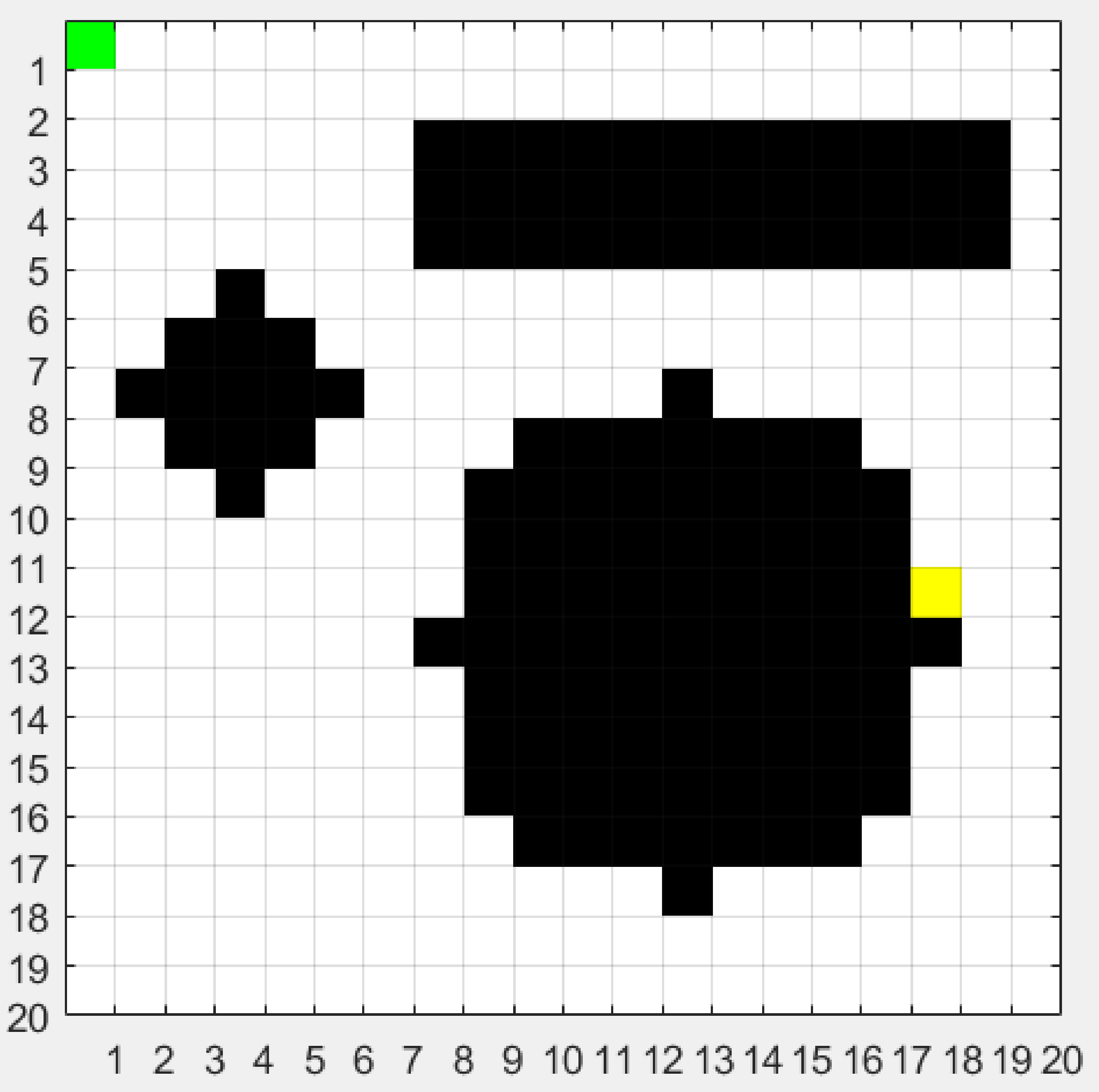
$S[2] = 1, Dist[2] = 2$

$S[3] = 1, Dist[3] = 4$

$S[4] = 1, Dist[4] = 6$

$S[5] = 1, Dist[5] = 15$

$S[6] = 1, Dist[6] = 9$


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20