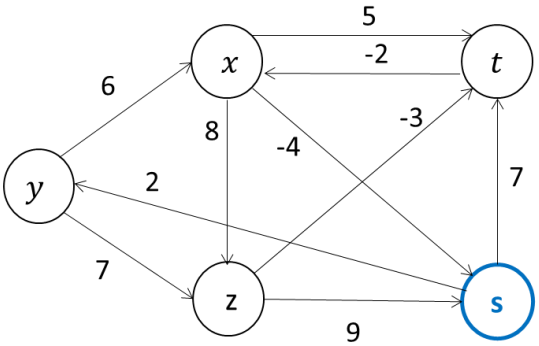
# Übung 11: Kürzeste Wege

### Aufgabe 1: Bellman-Ford

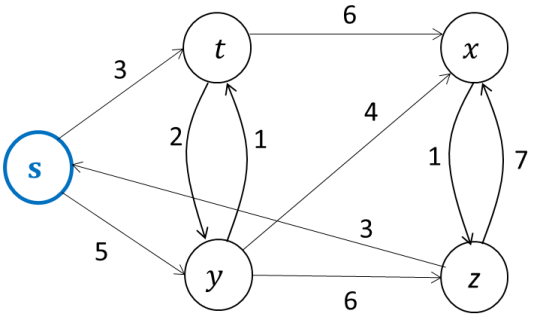
Wenden Sie auf den Graphen den Bellman-Ford Algorithmus der Vorlesung an. Der Startknoten sei ***s***. In jeder Runde werden zunächst alle Kanten besucht, die von *s* ausgehen, ***erst dann*** die von *t* ausgehenden Kanten, ***erst dann*** die von *x, y* und *z.* Die Adjazenzlisten seien zudem alphabetisch sortiert.




- Geben Sie **nur** an, welche Distanzen der Algorithmus ***nach der 1. Runde*** errechnet hat, also wenn alle Kanten ***genau einmal*** besucht wurden.
- Markieren Sie den ***Kürzeste-Wege-Baum*** zu diesem Zeitpunkt. Dieser wird bestimmt durch die Vorgänger $v.\pi$
- Wie viele weitere Runden sind notwendig, bis das finale Ergebnis feststeht?

### Aufgabe 2: Dijkstra

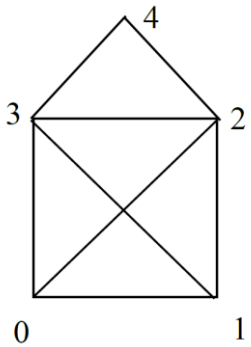
Wenden Sie auf die Abbildung den Dijkstra-Algorithmus der Vorlesung an. Der Startknoten sei ***s***.




- Geben Sie die Distanzen *d* für jeden Knoten ***am Ende jeder Iteration*** der while-Schleife an.
- Zeichnen Sie den Kürzeste-Wege-Baum!
- *Zum Knobeln*: Suchen Sie ein Beispiel für einen ***zyklenfreien*** Graphen mit negativen Gewichten, bei der der Dijkstra-Algorithmus der Vorlesung nicht funktioniert.

### Aufgabe 3: Haus des Nikolaus - Backtracking

Schreiben Sie ein rekursives Programm, das alle Möglichkeiten ausgibt, wie man das Haus des Nikolaus zeichnen kann, und zwar ohne mit dem Stift abzusetzen und ohne eine Linie zweimal zu durchlaufen. ***Starten Sie immer vom Knoten 0***. Wie viele Lösungen gibt es?




Modellieren Sie das Problem ungerichteten Graphen. Die Knoten sind durchnummeriert (0, 1, …, 4). Im vorgegebenen Code ist der Graph als Adjazenzmatrix vorgegeben, wobei `adjMatrix[i][j]=true` bedeutet, dass zwischen `i` und `j` eine Kante existiert.

**Beachten Sie:**

- Die Lösung wird in der ArrayList **`sol`** repräsentiert. Bsp: `sol`=[0 2 4 3 2 1 0 3 1] bedeutet, dass erst Kante (0,2), dann (2,4), dann (4,3), usw. gezeichnet wird. Zunächst enthält `sol` nur den Startknoten 0, also `sol`=[0]. Die Gesamtlösung muss aus 9 Elementen (= 8 Kanten) bestehen.

- Verwenden Sie ***Rekursion!*** Ähnlich wie bei einer Tiefensuche wird versucht, die ArrayList `sol` wachsen zu lassen bis sie 9 Elemente enthält (= 8 Kanten, Endergebnis). Die rekursive Methode **`back()`** versucht stets das aktuelle Zwischenergebnis, das in `sol` gespeichert ist, zu erweitern.

Um den aktuellen Stand um 1 Kanten bzw. Knoten zu erweitern, geht die Methode `back()` zu allen Nachbarn des zuletzt hinzugefügten Knotens. Um zu verhindern, dass man eine Kante später nochmal durchläuft, wird die Kante zwischenzeitlich aus der Adjazenzmatrix gelöscht und beim Rückkehr aus der Rekursion wieder hinzugefügt.

- Es genügt nicht nur eine Lösung zu finden, ermitteln Sie ***alle*** Lösungen!

**Frohe Weihnachten und einen guten Start ins neue Jahr!**