Source Code 1. Implementierung: Spanning–Tree–Algorithmus.

```
def find_spanning_tree2(adj_matrix):
    """Algorithmus zur Konstruktion eines spannenden Baumes"""
    # adj_matrix sollte die Adjazenzmatrix des (einfachen) Graphen
    # sein: Eintrag 0 bedeutet, daß die entsprechende
    # Kante nicht existiert.
    dim = adj_matrix.shape[0]
    # Zu Beginn gibt es keine Kante im spannenden Wald ...
    list_of_used_edges = []
    # ... und jeder Knoten ist ein Singleton-Block; die Blöcke (Komponenten)
    # werden mit einer _Funktion von beschränktem Wachstum_ codiert; zu
    # Beginn also durch (0,1,...,dim-1):
    fn_restricted_growth = np.arange(dim, dtype=int)
    # Die Kanten des Graphen entsprechen den nicht-Null-Einträgen _strikt
    # oberhalb_ der Hauptdiagonale; wir extrahieren sie aus der
    # Adjazenzmatrix.
    # Zuerst bestimmen wir einmal die _Anzahl_ der Kanten; mithilfe von
    # numpy-Funktionen: triu(m,j) liefert die Dreiecksmatrix "j Diagonalen
    # oberhalb der Hauptdiagonalen von m", und count_nonzero(matrix) zählt
    # die Einträge in matrix, die nicht Null sind:
    nof_edges = np.count_nonzero(np.triu(adj_matrix,1))

    # Vorbereitung von arrays, die die Kanten-Informationen aufnehmen
    # sollen:
    list_of_edges = np.zeros(2*nof_edges, dtype=int).reshape(nof_edges, 2)
    # "Auslesen" der Kanten aus der Adjazenzmatrix:
    k = 0
    for i in range(dim):
        for j in range(i+1,dim):
            entry = adj_matrix[i][j]
            if entry:
                list_of_edges[k][0] = i
                list_of_edges[k][1] = j
                k+= 1

    # Jetzt der eigentliche Algorithmus:
    for (a,b) in list_of_edges:
        # Die Funktion von beschränktem Gewicht codiert, zu welchen
        # Blöcken (Komponenten) die Endpunkte a, b gehören:
        block_no_1 = fn_restricted_growth[a]
        block_no_2 = fn_restricted_growth[b]
        # Verbindet Kante (a,b) zwei Knoten in derselben Komponente?
        if block_no_1 == block_no_2:
            # Ja: Überspringe den Rest der Schleife!
            continue
        # Implicit else:
        # Nein, diese Kante verbindet zwei _verschiedene_ Komponenten!
        # Wir verwenden sie also für den spannenden Wald:
        list_of_used_edges+= [[a,b]]
```

```
            # Die neu verbundenen Komponenten werden nun zu einer neuen
            # Komponenten "verschmolzen", das müssen wir in
            # fn_restricted_growth abbilden:
            if block_no_2 < block_no_1:
                dummy = block_no_2
                block_no_2 = block_no_1
                block_no_1 = dummy
            # Nun gilt jedenfalls block_no_1 < block_no_2
            # Wieder numpy-Funktionalität: "fn_restricted_growth==block_no_2"
            # liefert jenes Array von _indices_ i, für die
            # fn_restricted_growth[i]==block_no_2 gilt; die werden dann mit
            # block_no_1 überschrieben:
            fn_restricted_growth[
                np.nonzero(fn_restricted_growth==block_no_2)
            ] = block_no_1
            # Mit derselben numpy-Technik: Alle Funktionswerte von
            # fn_restricted_growth, die größer als block_no_2 sind, werden
            # um 1 vermindert:
            for k in range(block_no_2+1,np.max(fn_restricted_growth)+1):
                fn_restricted_growth[np.nonzero(fn_restricted_growth==k)]-=1

    # Rückgabe: Kanten des minimalen spannenden Waldes und
    # Zusammenhangskomponenten (als Partition der Knotenmenge;
    # codiert als Funktion von beschränktem Wachstum):
    return list_of_used_edges, fn_restricted_growth
```