این مسئله بسیار به مسئله جریان بیشینه شبیه است و با استفاده از تکنیک تبدیل مسئله می‌توان آن‌را حل کرد. روند تبدیل این مسئله بدین صورت است که به جای هر راس این گراف دو راس در گراف جدید قرار می‌دهیم به طوری که تمام یال‌هایی که به راس v وارد می‌شوند را به v_{in} وارد می‌کنیم و تمام راس‌هایی که از v خارج می‌شوند را از v_{out} خارج می‌کنیم و یک یال بین این دو راس قرار می‌دهیم که ظرفیت آن برابر با ظرفیت v است و با حل مسئله جدید، جواب مسئله اصلی را پیدا می‌کنیم.

ابتدا نحوه‌ی نمایش گراف را بصورتی انتخاب می‌کنیم که روند حل مسئله را برای ما راحت‌تر کند. اگر برای نمایش گراف از ماتریس استفاده کنیم، پس از تبدیل درایه‌های بسیار زیادی از ماتریس صفر خواهند بود که سرعت برنامه را کاهش خواهد و تبدیل مسئله و تبدیل نتیجه الگوریتم به جواب را سخت‌تر خواهد کرد بنابراین استفاده از لیست مجاورت و ذخیره اطلاعتی اضافی در آن بسیار آسان‌تر خواهد بود البته در اینجا برای سادگی کار و ذخیره این اطلاعت اضافی از $dictionary$ استفاده شده است. ساختار نمونه‌ای از یک گراف در $python$ به صورت زیر خواهد بود:

```
g = {
    1: {"n": {2: (2, 0), 4: (3, 0)}, "f": (4, 0)},
    2: {"n": {3: (5, 0), 5: (3, 0)}, "f": (10, 0)},
    3: {"n": {6: (2, 0)}, "f": (3, 0)},
    4: {"n": {3: (1, 0)}, "f": (2, 0)},
    5: {"n": {6: (4, 0)}, "f": (3, 0)},
    6: {"n": {}, "f": (12, 0)},
}
```

در اینجا به هر راس همسایه‌های آن (n) و جریانی که از آن راس می‌گذرد (f) را نسبت می‌دهیم. هر یک از این دوتایی‌ها به صورت $(capacity, flow)$ هستند.

برای پیدا کردن جریان بیشینه یک گراف از الگوریتم $ford-fulkerson$ استفاده می‌کنیم. پیاده‌سازی این الگوریتم با در نظر گرفتن ساختار گراف به صورت زیر خواهد بود:

```
def ford_fulkerson(graph, source, destination):
    while (path := bfs_path(graph, source, destination)) is not None:
        path_flow = math.inf
        for i in range(0, len(path) - 1):
            capacity, flow = graph[path[i]]["n"][path[i + 1]]
            path_flow = min(path_flow, capacity - flow)
        for i in range(0, len(path) - 1):
            u, v = path[i], path[i + 1]
            capacity, flow = graph[u]["n"][v]
            graph[u]["n"][v] = (capacity, flow + path_flow)
            capacity, flow = graph[v]["n"].get(u, (0, 0))
            graph[v]["n"][u] = (capacity, flow - path_flow)
    for node in graph.keys():
        for key, value in list(graph[node]["n"].items()):
            if value[1] < 0:
                del graph[node]["n"][key]
```

الگوریتم بدین‌صورت عمل خواهد کرد که در صورت وجود مسیری جریان‌افزا که آن را از bfs_path می‌گیرد مقدرا جریانی که می‌تواند از این مسیر عبور کند را که برابر است با کم ظرفیت‌ترین یال یا خط ارتباطی (با در نظر گرفتن جریانی که در حال حاظر دارند) را پیدا می‌کند و جریان گذرنده از این یال را بروز می‌کند. ذکر این نکته ضروری است که در این الگوریتم اگر از یک یال جهتدار جریانی عبور دهیم، می‌توانیم به همین مقدار از آن کم‌کنیم و جریان را برگردانیم به همین دلیل مقدار جریان یال برگشتی‌را از مقدار جریان کم‌میکنیم و اگر وجود نداشته باشد آن را ایجاد می‌کنیم. در انتها یال‌هایی که جریان گذرنده از آن‌ها منفی است را حذف می‌کنیم تا خروجی الگوریتم درست باشد.

در اینجا به الگوریتم bfs_path احتیاج داشتیم که پیاده سازی آن بدین صورت خواهد بود:

```python
def bfs_path(graph, source, destination):
    queue = deque([source])
    predecessors = {source: None}
    visited = set([source])

    while queue:
        node = queue.popleft()
        if node == destination:
            path = []
            while node is not None:
                path.append(node)
                node = predecessors[node]
            path.reverse()
            return path
        for neighbor in graph[node]["n"]:
            capacity, flow = graph[node]["n"][neighbor]
            if neighbor not in visited and flow < capacity:
                visited.add(neighbor)
                predecessors[neighbor] = node
                queue.append(neighbor)

    return None
```

در اینجا الگوریتم همانند پیمایش سطحی عمل می‌کند با این تفاوت که علاوه بر شرط دیده نشدن راس، این که جریان گذرنده از یال کمتر از ظرفیت آن باشد را نیز بررسی می‌کند. برای اینکه مسیر پیدا شده توسط این الگوریتم را بیابیم، پدر یا راسي که از آن به راس دیگر رفته‌ایم را ذخیره می‌کنیم تا بتوانیم از روی آن‌ها مسیر را برگردانیم.

حال پس از پیاده‌سازی الگوریتم‌های اصلی، الگوریتم‌های تبدیل را طراحی می‌کنیم:

```python
def transfer_graph(graph):
    i = 0
    new_graph = {}
    for node in graph:
        i -= 1
        in_node = {"n": {i: graph[node]["f"]}, "out": i, "in": True}
        out_node = {"n": graph[node]["n"]}
        new_graph[node] = in_node
        new_graph[i] = out_node
    return new_graph, i
```

برای تبدیل گراف اصلی به گراف دلخواه به ازای هر راس دوراس جدید ایجاد می‌کنیم. نام یکی از آنها را که راس v_{in} است با نام راس اصلی یکسان قرار می‌دهیم تا مشکلی برای پیمایش گراف به وجود نیاید چرا که در صورت ایجاد تغییر یا قراردادن نامی دلخواه باید تمام همسایه‌ها را در همه‌ی راس‌ها تغییر دهیم. این راس تنها به یک راس متصل است که آن راس v_{out} خواهد بود که تمام همسایه‌های راس اصلی را به آن نسبت می‌دهیم. برای برگرداندن این گراف به گراف اصلی، یکی از این راس‌ها را علامتگذاری می‌کنیم و نام راس دیگر را به این راس می‌دهیم. البته باید توجه کنیم که نام راس‌های v_{out} از منفی ۱ شروع می‌شود و نباید گراف اصلی از اعداد منفی برای نام گذاری راس‌های اصلی استفاده کرده باشد. در انتها گراف جدید و نام آخرین راس آن‌را برمی‌گردانیم.

```python
def revert_graph_transform(graph):
    og_graph = {}
```

```
for node in graph:
    if graph[node].get("in", False):
        out_node = graph[node]["out"]
        og_graph[node] = {
            "n": graph[out_node]["n"],
            "f": graph[node]["n"][out_node],
        }
return og_graph
```

پس از اجرای الگوریتم، با استفاده از این تابع جواب نهایی مسئله اصلی را پیدا می‌کنیم.