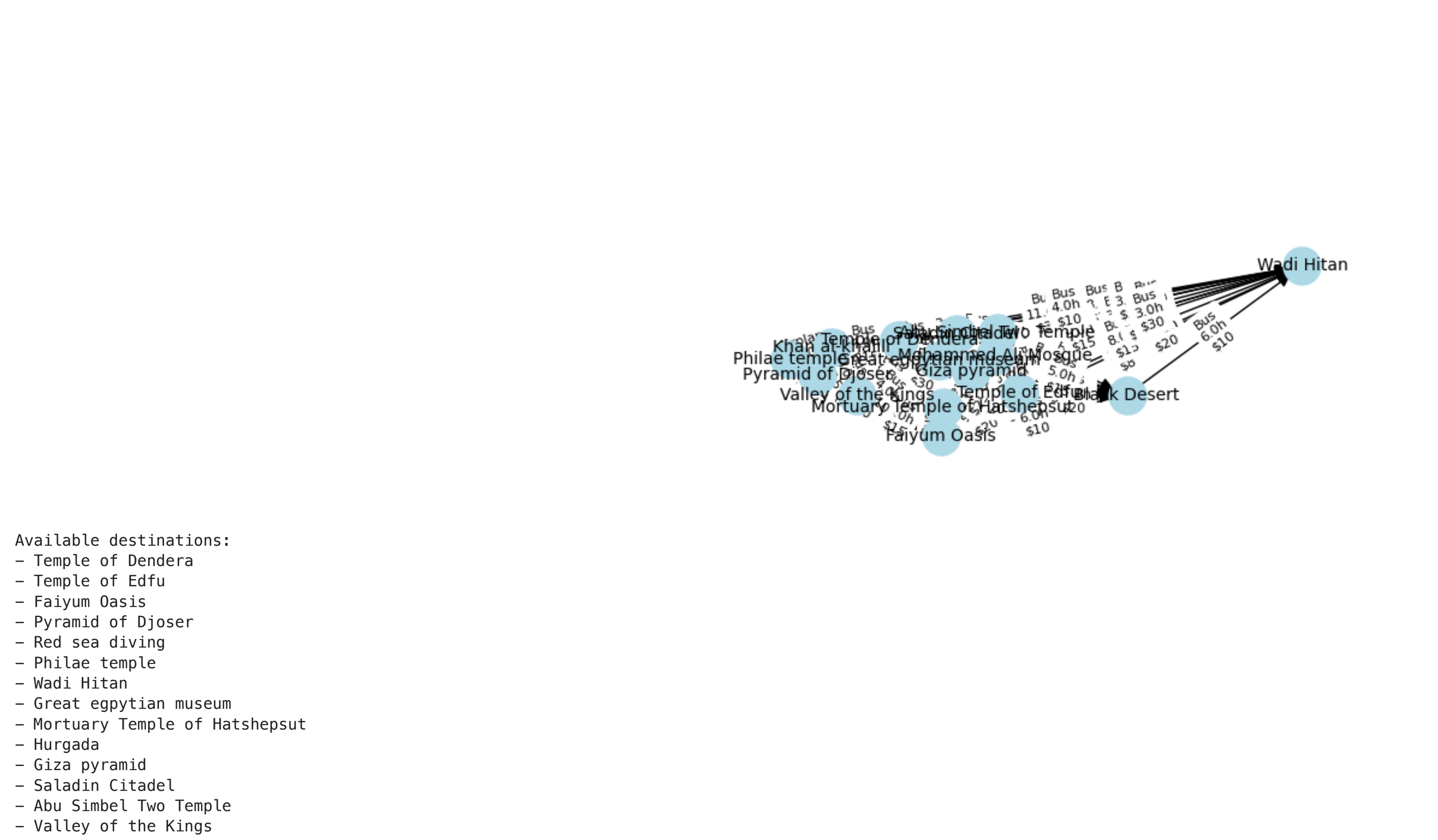
In [1]:

```python
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
import random
from collections import defaultdict
import itertools

class TravelPlanner:
    def __init__(self, excel_file):
        """Initialize travel planner with data from Excel file"""
        self.df = pd.read_excel(excel_file, sheet_name='Transportation List')
        self.graph = nx.DiGraph()
        self.build_network()

    def build_network(self):
        """Build network from Excel data"""
        # Add all places as nodes
        places = set(self.df['Place'].unique()) | set(self.df['To'].unique())
        for place in places:
            self.graph.add_node(place)

        # Add connections as edges with attributes
        for _, row in self.df.iterrows():
            self.graph.add_edge(
                row['Place'],
                row['To'],
                transport=row['Transportation'],
                period=row['Transportation Period(Hour)'],
                price=row['Price($)']
            )

    def visualize_network(self):
        """Visualize the travel network"""
        plt.figure(figsize=(12, 8))
        pos = nx.spring_layout(self.graph, seed=42)  # Fixed seed for consistent layout

        # Draw nodes
        nx.draw_networkx_nodes(self.graph, pos, node_size=500, node_color='lightblue')

        # Draw edges
        nx.draw_networkx_edges(self.graph, pos, arrowsize=15)

        # Add labels
        nx.draw_networkx_labels(self.graph, pos, font_size=10)

        # Add edge labels with transport, period and price
        edge_labels = {(u, v): f"{d['transport']}\n{d['period']}h\n${d['price']}"
                       for u, v, d in self.graph.edges(data=True)}
        nx.draw_networkx_edge_labels(self.graph, pos, edge_labels=edge_labels, font_size=8)

        plt.title("Travel Network Visualization")
        plt.axis('off')
        plt.tight_layout()
        plt.savefig('travel_network.png')  # Save the visualization
        plt.show()

    def get_all_places(self):
        """Get all places in the network"""
        return list(self.graph.nodes())

    def find_shortest_path(self, start, end, metric='period'):
        """Find shortest path between two places based on specified metric"""
        if metric == 'period':
            weight = 'period'
        elif metric == 'price':
            weight = 'price'
        else:
            raise ValueError("Metric must be either 'transport' or 'price'")

        try:
            path = nx.shortest_path(self.graph, start, end, weight=weight)
            path_edges = list(zip(path[:-1], path[1:]))
            total_weight = sum(self.graph[u][v][weight] for u, v in path_edges)
            transport_methods = [self.graph[u][v]['transport'] for u, v in path_edges]

            return {
                'path': path,
                f'total_{metric}': total_weight,
                'transport': transport_methods
            }
        except (nx.NetworkXNoPath, nx.NodeNotFound):
            return None

    def find_all_paths(self, start, end, cutoff=None):
        """Find all simple paths from start to end"""
        try:
            paths = list(nx.all_simple_paths(self.graph, start, end, cutoff=cutoff))
            return paths
        except (nx.NetworkXNoPath, nx.NodeNotFound):
            return []

    def calculate_path_metrics(self, path):
        """Calculate metrics for a given path"""
        if len(path) < 2:
            return None

        path_edges = list(zip(path[:-1], path[1:]))
        total_period = sum(self.graph[u][v]['period'] for u, v in path_edges)
        total_price = sum(self.graph[u][v]['price'] for u, v in path_edges)
        transport_methods = [self.graph[u][v]['transport'] for u, v in path_edges]

        return {
            'path': path,
            'total_period': total_period,
            'total_price': total_price,
            'transport_methods': transport_methods
        }

    def get_optimal_travel_plans(self, start, end, cutoff=7, top_n=7):
        """Find optimal travel plans based on time and cost"""
        all_paths = self.find_all_paths(start, end, cutoff)

        if not all_paths:
            return []

        # Calculate metrics for each path
        path_metrics = [self.calculate_path_metrics(path) for path in all_paths]

        # Find fastest plans
        fastest_plans = sorted(path_metrics, key=lambda x: x['total_period'])[:top_n]

        # Find cheapest plans
        cheapest_plans = sorted(path_metrics, key=lambda x: x['total_price'])[:top_n]

        # Find balanced plans (normalize and combine time and cost)
        max_period = max(plan['total_period'] for plan in path_metrics)
        max_price = max(plan['total_price'] for plan in path_metrics)

        for plan in path_metrics:
            plan['normalized_score'] = (plan['total_period'] / max_period) + (plan['total_price'] / max_price)

        balanced_plans = sorted(path_metrics, key=lambda x: x['normalized_score'])[:top_n]

        return {
            'fastest': fastest_plans,
            'cheapest': cheapest_plans,
            'balanced': balanced_plans
        }

    def generate_multi_city_plans(self, must_visit, optional_visit=None, start=None, end=None, max_cities=8):
        """Generate multi-city travel plans, ensuring must_visit cities are included"""
        if optional_visit is None:
            optional_visit = []

        # If start/end not specified, use first must_visit city
        if start is None:
            start = must_visit[0] if must_visit else None
        if end is None:
            end = must_visit[-1] if must_visit else start

        if start is None or end is None:
            return []

        # Ensure start and end are in must_visit
        must_visit_set = set(must_visit)
        if start not in must_visit_set:
            must_visit = [start] + must_visit
        if end not in must_visit_set and end != start:
            must_visit = must_visit + [end]

        # Generate all possible city combinations including must_visit
        must_visit_set = set(must_visit)
        remaining_cities = set(self.get_all_places()) - must_visit_set
        optional_cities = set(optional_visit) & remaining_cities
        other_cities = remaining_cities - optional_cities

        # Limit max additional cities
        max_additional = max_cities - len(must_visit)
        if max_additional < 0:
            max_additional = 0

        plans = []

        # Prioritize optional_visit cities
        for n in range(min(max_additional + 1, len(optional_cities) + 1)):
            for opt_combo in itertools.combinations(optional_cities, n):
                remaining = max_additional - n
                if remaining > 0:
                    for other_combo in itertools.combinations(other_cities, min(remaining, len(other_cities))):
                        city_set = list(must_visit_set | set(opt_combo) | set(other_combo))
                        plans.extend(self._evaluate_city_combination(city_set, start, end))
                else:
                    city_set = list(must_visit_set | set(opt_combo))
                    plans.extend(self._evaluate_city_combination(city_set, start, end))

        # Sort plans by a balanced score
        for plan in plans:
            plan['balanced_score'] = plan['total_period'] * 0.5 + plan['total_price'] * 0.5

        return sorted(plans, key=lambda x: x['balanced_score'])[:10]

    def _evaluate_city_combination(self, cities, start, end):
        """Evaluate a combination of cities for travel planning"""
        # Filter to only cities in our network
        cities = [city for city in cities if city in self.graph]

        if not cities or start not in cities or end not in cities:
            return []

        # Generate permutations that start with start and end with end
        cities_to_permute = [city for city in cities if city != start and city != end]
        permutations = list(itertools.permutations(cities_to_permute))

        plans = []
        for perm in permutations:
            path = [start] + list(perm) + [end]

            # Check if this path is valid (all connections exist)
            valid = True
            for i in range(len(path) - 1):
                if not self.graph.has_edge(path[i], path[i+1]):
                    valid = False
                    break

            if valid:
                metrics = self.calculate_path_metrics(path)
                if metrics:
                    plans.append(metrics)

        return plans
    def find_paths_with_constraints(self, start, end, max_cost=None, max_time=None, cutoff=None):
        """Find all paths that satisfy cost and time constraints"""
        all_paths = self.find_all_paths(start, end, cutoff)
        valid_paths = []

        for path in all_paths:
            metrics = self.calculate_path_metrics(path)
            if metrics is None:
                continue

            # Check if path satisfies both constraints
            is_valid = True
            if max_cost is not None and metrics['total_price'] > max_cost:
                is_valid = False
            if max_time is not None and metrics['total_period'] > max_time:
                is_valid = False

            if is_valid:
                valid_paths.append(metrics)

        return valid_paths

    def get_optimal_travel_plans_with_constraints(self, start, end, max_cost=None, max_time=None, cutoff=7, top_n=7):
        """Find optimal travel plans based on time and cost, with constraints"""
        valid_paths = self.find_paths_with_constraints(start, end, max_cost, max_time, cutoff)

        if not valid_paths:
            return {
                'fastest': [],
                'cheapest': [],
                'balanced': [],
                'message': f"No paths found within constraints (Max cost: ${max_cost}, Max time: {max_time} hours)"
            }

        # Find fastest plans
        fastest_plans = sorted(valid_paths, key=lambda x: x['total_period'])[:top_n]

        # Find cheapest plans
        cheapest_plans = sorted(valid_paths, key=lambda x: x['total_price'])[:top_n]

        # Find balanced plans
        max_period = max(plan['total_period'] for plan in valid_paths)
        max_price = max(plan['total_price'] for plan in valid_paths)

        for plan in valid_paths:
            plan['normalized_score'] = (plan['total_period'] / max_period) + (plan['total_price'] / max_price)

        balanced_plans = sorted(valid_paths, key=lambda x: x['normalized_score'])[:top_n]

        return {
            'fastest': fastest_plans,
            'cheapest': cheapest_plans,
            'balanced': balanced_plans,
            'message': f"Found paths within constraints (Max cost: ${max_cost}, Max time: {max_time} hours)"
        }

    def generate_multi_city_plans_with_constraints(self, must_visit, max_cost=None, max_time=None,
                                                   optional_visit=None, start=None, end=None, max_cities=8):
        """Generate multi-city travel plans with cost and time constraints"""
        plans = self.generate_multi_city_plans(must_visit, optional_visit, start, end, max_cities)

        # Filter plans based on constraints
        valid_plans = []
        for plan in plans:
            if max_cost is not None and plan['total_price'] > max_cost:
                continue
            if max_time is not None and plan['total_period'] > max_time:
                continue
            valid_plans.append(plan)

        return valid_plans

def analyze_travel_options(excel_file):
    """Analyze travel options from Excel file"""
    planner = TravelPlanner(excel_file)

    # Visualize the network
    planner.visualize_network()

    # Display all places
    places = planner.get_all_places()
    print("Available destinations:")
    for place in places:
        print(f"- {place}")

    # Sample optimal travel plans
    if len(places) >= 2:
        start = places[0]
        end = places[-1]
        print(f"\nSample travel plans from {start} to {end}:")
        plans = planner.get_optimal_travel_plans(start, end)

        if plans and plans.get('fastest'):
            print("\nFastest Travel Plans:")
            for i, plan in enumerate(plans['fastest'], 1):
                print(f"{i}. Route: {' -> '.join(plan['path'])}")
                print(f"   Total time: {plan['total_period']} hours")
                print(f"   Total cost: ${plan['total_price']}")
                print(f"   Transportation: {' -> '.join(plan['transport_methods'])}")

        if plans and plans.get('cheapest'):
            print("\nCheapest Travel Plans:")
            for i, plan in enumerate(plans['cheapest'], 1):
                print(f"{i}. Route: {' -> '.join(plan['path'])}")
                print(f"   Total time: {plan['total_period']} hours")
                print(f"   Total cost: ${plan['total_price']}")
                print(f"   Transportation: {' -> '.join(plan['transport_methods'])}")

        # Generate a multi-city plan example
        print("\nExample Multi-City Travel Plan:")
        must_visit = [places[0], places[-1]]
        if len(places) > 3:
            must_visit.append(places[len(places)//2])

        multi_plans = planner.generate_multi_city_plans(must_visit)
        if multi_plans:
            plan = multi_plans[0]
            print(f"Route: {' -> '.join(plan['path'])}")
            print(f"Total time: {plan['total_period']} hours")
            print(f"Total cost: ${plan['total_price']}")
            print(f"Transportation: {' -> '.join(plan['transport_methods'])}")

    return planner

def interactive_planner(excel_file):
    """Interactive travel planner function"""
    planner = TravelPlanner(excel_file)
    places = planner.get_all_places()

    print("\n=== Interactive Travel Planner ===")
    print("Available destinations:")
    for i, place in enumerate(places, 1):
        print(f"{i}. {place}")

    try:
        print("\nSelect starting point (enter number):")
        start_idx = int(input()) - 1
        start = places[start_idx]

        print("Select destination (enter number):")
        end_idx = int(input()) - 1
        end = places[end_idx]

        # Add max cost constraint here
        print("\nEnter maximum budget in $ (press enter for no limit):")
        max_cost_input = input().strip()
        max_cost = float(max_cost_input) if max_cost_input else None

        # Add max time constraint here
        print("Enter maximum travel time in hours (press enter for no limit):")
        max_time_input = input().strip()
        max_time = float(max_time_input) if max_time_input else None

        print("Must visit locations (comma-separated numbers, press enter to skip):")
        must_visit_input = input()
        must_visit = []
        if must_visit_input.strip():
            must_visit_idxs = [int(x.strip()) - 1 for x in must_visit_input.split(',')]
            must_visit = [places[idx] for idx in must_visit_idxs if 0 <= idx < len(places)]

        print("Planning your optimal routes...")

        # Get regular plans
        plans = planner.get_optimal_travel_plans_with_constraints(
            start=start,
            end=end,
            max_cost=max_cost,
            max_time=max_time
        )

        # Show fastest plans
        if plans['fastest']:
            print("\n--- Fastest Travel Plans (Within Constraints) ---")
            for i, plan in enumerate(plans['fastest'], 1):
                print(f"{i}. Route: {' -> '.join(plan['path'])}")
                print(f"   Total time: {plan['total_period']} hours")
                print(f"   Total cost: ${plan['total_price']}")
                print(f"   Transportation: {' -> '.join(plan['transport_methods'])}")
        else:
            print("\nNo fastest plans found within the constraints.")

        # Show cheapest plans
        if plans['cheapest']:
            print("\n--- Cheapest Travel Plans (Within Constraints) ---")
            for i, plan in enumerate(plans['cheapest'], 1):
                print(f"{i}. Route: {' -> '.join(plan['path'])}")
                print(f"   Total time: {plan['total_period']} hours")
                print(f"   Total cost: ${plan['total_price']}")
                print(f"   Transportation: {' -> '.join(plan['transport_methods'])}")
        else:
            print("\nNo cheapest plans found within the constraints.")

        # Show multi-city plans
        if must_visit:
            print("\n--- Multi-City Travel Plans (Including Must-Visit Locations) ---")
            multi_plans = planner.generate_multi_city_plans_with_constraints(
                must_visit=must_visit,
                max_cost=max_cost,
                max_time=max_time,
                start=start,
                end=end
            )
            if multi_plans:
                for i, plan in enumerate(multi_plans[:5], 1):
                    print(f"{i}. Route: {' -> '.join(plan['path'])}")
                    print(f"   Total time: {plan['total_period']} hours")
                    print(f"   Total cost: ${plan['total_price']}")
                    print(f"   Transportation: {' -> '.join(plan['transport_methods'])}")
            else:
                print("No multi-city plans found within the specified constraints.")

    except (ValueError, IndexError) as e:
        print(f"Error in input: {e}")
        print("Returning to analysis mode...")

    return planner

# Main function
if __name__ == "__main__":
    import sys

    if len(sys.argv) > 1:
        excel_file = '/Users/evehuang/Downloads/Egypt.xlsx'
    else:
        excel_file = '/Users/evehuang/Downloads/Egypt.xlsx'

    try:
        # First analyze general options
        planner = analyze_travel_options(excel_file)
        # Then allow interactive planning
        print("\nWould you like to use the interactive planner? (y/n)")
        if input().lower().startswith('y'):
            interactive_planner(excel_file)

    except Exception as e:
        print(f"Error: {e}")
        print("\nExpected Excel file format:")
        print("Columns: place, to, transportation method, transportation period, price")
        print("Each row represents a connection between two places")
```




```
Available destinations:
- Temple of Dendera
- Temple of Edfu
- Faiyum Oasis
- Pyramid of Djoser
- Red sea diving
- Philae temple
- Wadi Hitan
- Great egyptian museum
- Mortuary Temple of Hatshepsut
- Hurgada
- Giza pyramid
- Saladin Citadel
- Abu Simbel Two Temple
- Valley of the Kings
- Black Desert
- Mohammed Ali Mosque
- Khan al-khalili

Sample travel plans from Temple of Dendera to Khan al-khalili:

Example Multi-City Travel Plan:

Would you like to use the interactive planner? (y/n)
=== Interactive Travel Planner ===
Available destinations:
1. Temple of Dendera
2. Temple of Edfu
3. Faiyum Oasis
4. Pyramid of Djoser
5. Red sea diving
6. Philae temple
7. Wadi Hitan
8. Great egyptian museum
9. Mortuary Temple of Hatshepsut
10. Hurgada
11. Giza pyramid
12. Saladin Citadel
13. Abu Simbel Two Temple
14. Valley of the Kings
15. Black Desert
16. Mohammed Ali Mosque
17. Khan al-khalili

Select starting point (enter number):
Select destination (enter number):
Enter maximum budget in $ (press enter for no limit):
Enter maximum travel time in hours (press enter for no limit):
Must visit locations (comma-separated numbers, press enter to skip):
Planning your optimal routes...

--- Fastest Travel Plans (Within Constraints) ---
1. Route: Giza pyramid -> Khan al-khalili -> Philae temple
   Total time: 2.5 hours
   Total cost: $85
   Transportation: Taxi -> Airplane
2. Route: Giza pyramid -> Great egyptian museum -> Khan al-khalili -> Philae temple
   Total time: 2.66 hours
   Total cost: $87
   Transportation: Taxi -> Taxi -> Airplane
3. Route: Giza pyramid -> Mohammed Ali Mosque -> Khan al-khalili -> Philae temple
   Total time: 3.0 hours
   Total cost: $90
   Transportation: Taxi -> Taxi -> Airplane
4. Route: Giza pyramid -> Saladin Citadel -> Khan al-khalili -> Philae temple
   Total time: 3.0 hours
   Total cost: $90
   Transportation: Taxi -> Taxi -> Airplane
5. Route: Giza pyramid -> Great egyptian museum -> Mohammed Ali Mosque -> Khan al-khalili -> Philae temple
   Total time: 3.16 hours
   Total cost: $92
   Transportation: Taxi -> Taxi -> Taxi -> Airplane
6. Route: Giza pyramid -> Great egyptian museum -> Saladin Citadel -> Khan al-khalili -> Philae temple
   Total time: 3.16 hours
   Total cost: $92
   Transportation: Taxi -> Taxi -> Taxi -> Airplane
7. Route: Giza pyramid -> Mohammed Ali Mosque -> Saladin Citadel -> Khan al-khalili -> Philae temple
   Total time: 3.5 hours
   Total cost: $95
   Transportation: Taxi -> Taxi -> Taxi -> Airplane

--- Cheapest Travel Plans (Within Constraints) ---
1. Route: Giza pyramid -> Philae temple
   Total time: 12.0 hours
   Total cost: $30
   Transportation: Bus
2. Route: Giza pyramid -> Pyramid of Djoser -> Philae temple
   Total time: 11.6 hours
   Total cost: $31
   Transportation: Taxi -> Bus
3. Route: Giza pyramid -> Great egyptian museum -> Philae temple
   Total time: 12.16 hours
   Total cost: $32
   Transportation: Taxi -> Bus
4. Route: Giza pyramid -> Great egyptian museum -> Pyramid of Djoser -> Philae temple
   Total time: 11.76 hours
   Total cost: $33
   Transportation: Taxi -> Taxi -> Bus
5. Route: Giza pyramid -> Mohammed Ali Mosque -> Philae temple
   Total time: 12.5 hours
   Total cost: $35
   Transportation: Taxi -> Bus
6. Route: Giza pyramid -> Saladin Citadel -> Philae temple
   Total time: 12.5 hours
   Total cost: $35
   Transportation: Taxi -> Bus
7. Route: Giza pyramid -> Mohammed Ali Mosque -> Pyramid of Djoser -> Philae temple
   Total time: 12.1 hours
   Total cost: $36
   Transportation: Taxi -> Taxi -> Bus

--- Multi-City Travel Plans (Including Must-Visit Locations) ---
No multi-city plans found within the specified constraints.
```

In [ ]: