Die Lösungsidee:

Meine Lösungsidee zu der Aufgabe ähnelt sehr der Lösungsidee zur Aufgabe „Telepaartie“.
Bei dieser Aufgabe kann zunächst für jede Tankstelle sowie die Startposition kontrolliert werden, zu welchen Tankstellen man von dort aus gelangen kann sowie ob es möglich ist von diesem Punkt aus das Ziel zu erreichen. Anschließend probiert man alle möglichen Routen aus, die unter Beachtung der Tankgröße und des Verbrauchs möglich sind. Dies geschieht durch einen rekursiven Aufruf einer Methode, welche immer die derzeitige Position (Tankstelle) und die bis zu dieser Position vorangegangenen Positionen. Die Methode ruft dann für jede Position sich selbst erneut auf, wobei als aktuelle Position je eine der von der letzten Position aus erreichbaren Positionen übergeben wird. Dies geschieht, bis das Ziel erreicht wurde oder eine maximal Anzahl an angefahrenen Tankstellen überschritten wird. Diese maximale Anzahl ist zunächst ∞ und wird nach dem ersten Mal bei dem das Ziel erreicht wird auf die Zahl der zum Erreichen des Ziels benötigten Tankstellen gesetzt. Dieser maximale Wert wird jedes Mal reduziert, wenn ein Weg mit einer geringeren Anzahl an anzufahrenden Tankstellen gefunden wird.

Für beispielsweise den ersten Testfall funktioniert dieses Verfahren bereits ziemlich gut, doch für zum Beispiel den letzten Testfall ist dieses Verfahren ziemlich langsam, da von jeder Tankstelle aus gleich einige andere Tankstellen angefahren werden können. Dies sorgt für eine stark ansteigende Laufzeit, weshalb ich an dieser Stelle das Verfahren noch etwas variabler gestalten wollte. Statt alle Möglichkeiten auszuprobieren, sollen nur jene ausprobiert werden, die vermutlich der besten Möglichkeit zuzuordnen sind. Das heißt an dieser Stelle: da möglichst wenige Tankstellen angefahren werden sollen, ist es am sinnvollsten von jeder Tankstelle aus zu einer möglichst weit entfernten zu fahren, da sich so der Weg im Verhältnis zu den anzufahrenden Tankstellen am schnellsten verringert. Dies bedeutet wiederum, dass es möglich ist eine Obergrenze festzulegen, welche besagt, für wie viele erreichbare Tankstellen von jeder Position aus maximal ein rekursiver Aufruf geschehen soll. Wie sich bei den Testfällen zeigte, lassen sich so auch die Testfälle 2 und 5 in wenigen Sekunden bis ein paar Minuten näherungsweise ziemlich genau lösen, während die anderen Testfälle in dieser Zeit optimal gelöst werden können.

Für eine Obergrenze von 3 ist auf meinem Computer ein Ergebnis in weniger als einer Sekunden erreichbar, für eine Obergrenze von 8 in ungefähr 20 Minuten (für die Testfällen 2 und 5).

Für eine Obergrenze x werden also von einer Tankstelle mit zum Beispiel 2x erreichbaren, nachfolgenden Tankstellen nur die weiter entfernt liegenden x Tankstellen ausprobiert. Von einer Tankstelle mit nur zum Beispiel x-1 erreichbaren, nachfolgenden Tankstellen werden folglich weiterhin alle erreichbaren Tankstellen ausprobiert. Ist die Obergrenze groß genug kann jedes Beispiel optimal gelöst werden. Es muss nur eine ausreichend hohe Obergrenze für diese Testfälle gesetzt werden - zum Beispiel reicht 42 in jedem Fall für die Testfälle 1, 3 und 4. Soll nun hingegen in jedem Fall ein Ergebnis in wenigen Sekunden gefunden werden und nicht erst nach etlichen Tagen im schlechtesten Fall, so kann einfach die Obergrenze dementsprechend nach unten gesetzt werden.

Wenn das Ergebnis ansonsten womöglich erst errechnet werden könnte, wenn man bereits verstorben ist, so bringt die Ersparnis von ein paar Cent oder Tankstellen schließlich auch nichts mehr.

Die Umsetzung mit Quelltext:

Die Umsetzung der Lösungsidee erfolgt in meiner Bearbeitung in der Programmiersprache Ruby. Das Programm liegt im Ordner „Urlaubsfahrt“ und stellt zeitgleich Quelltext und ausführbare Datei dar. Ich habe das Programm unter macOS 10.14.6 und der Ruby Version 2.3.7p456 (2018-03-28 revision 63024) [universal.x86_64-darwin18] getestet.

In den ersten 31 Zeilen des Programms findet das Einlesen einer Datei sowie das Festlegen der zuvor beschriebenen Obergrenze statt. Die Variablen Namen entsprechen denen der Materialseite dieser Runde. Dabei ist anzumerken, dass sich in dem Array „$d“, neben den Kilometerzahlen für die verschiedenen Tankstellen, sich auch die Kilometerzahl des Startes (also *0*) und die des Ziels (also „$l“), in meiner Implementation, befinden. In dem Array „$p“ befindet sich an erster und letzter Stelle zudem zusätzlich der Wert ∞. Dies verhindert im restlichen Verlauf, dass an diesen Positionen getankt werden kann.

Von Zeile 107 bis 130 befinden sich der Aufruf des Ganzen und die Ausgabe der Ergebnisse. Dabei wird der günstigste Weg aus allen gefundenen Wegen mit optimal geringer Anzahl an Tankstellen ausgewählt und ausgegeben. Das hier ausgegebene Ergebnis sollte für die Testfälle 1, 3 und 4 optimal sein, während es für die Testfälle 2 und 5 zumindest zu einer akzeptablen Annäherung an das optimale Ergebnis kommt. Je größer die Obergrenze gewählt wird, umso besser wird bei diesen Testfällen das Ergebnis, wie ich später noch an verschiedenen Ausgaben zeigen werde.

Die Zeilen 33 bis 105 beinhalten das Herzstück des Programmes. Auf diese Zeilen werde ich im Weiteren genauer eingehen:

```
def erreichbare_tankstellen(pos_ind)
    fuellung = pos_ind == 0 ? $f.to_f : $g.to_f
    erreichbar = $d[pos_ind].to_f + fuellung / $v * 100.0
    tankstellen = Array.new
    pos_ind += 1
    while(pos_ind<=$z and $d[pos_ind] <= erreichbar) do
        if($d[pos_ind] <= erreichbar) then tankstellen << pos_ind end
        pos_ind += 1
    end
    return tankstellen
end
```

Diese Methode berechnet die Indizes aller Tankstellen, welche von der Tankstelle mit dem Index „pos_ind“ erreichbar sind und gibt diese Indizes in einem Array zurück. Dabei wird stets die derzeitige Position in Kilometern mit den mit einer vollen Tankladung fahrbaren Kilometer addiert. Anschließend wird für alle Tankstellen kontrolliert, ob sie weniger oder gleich weit vom Start aus entfernt liegen, wie das Auto von der derzeitigen Tankstelle aus nach Aufbrauchen einer weiteren Tankladung gefahren sein könnte. Diese Tankstellen gelten somit als erreichbar. Für den Sonderfall, dass „pos_ind“ *0* ist, findet diese Kontrolle stattdessen mit der Distanz statt, die mit der Startfüllung aus erreichbar ist, da hier keine Tankstelle ist und somit der Tank nicht vollständig gefüllt werden kann, wenn er das noch nicht ist.

Die nachfolgende Methode „test“ hat die Funktion, Wege mit möglichst wenigen Tankstopps zu finden. Alle gefundenen Wege mit gleicher Zahl an Tankstopps werden dabei in dem Array „$tankstellen“ gespeichert. So lässt sich am Ende der Weg auswählen, der zugleich die geringste mögliche Anzahl an Tankstopps wie auch möglichst geringe Tankkosten aufweist.

```
$tankstellen = Array.new
$shortest = Float::INFINITY
```

Die Variable „$shortest“ beinhaltet die Information, wie viele Tankstellen auf dem bisher besten gefundenen Weg, also mit möglichst wenigen Tankstopps, angefahren werden müssen. Zunächst deshalb ∞, da im ersten Durchlauf noch kein Weg gefunden wurde und somit „$shortest“ als Abbruchbedingung nicht vorzeitig alles abbrechen kann.

```
def test(ind, vorherige_tankstellen)
  if $d[ind] == $l then
    if(vorherige_tankstellen.length < $shortest)
      $shortest = vorherige_tankstellen.length
      $tankstellen.clear
      $tankstellen.push(vorherige_tankstellen + [ind])
    elsif(vorherige_tankstellen.length == $shortest)
      $tankstellen.push(vorherige_tankstellen + [ind])
    end
```

Als erstes wird in jedem Aufruf der Methode kontrolliert, ob das Ziel erreicht wurde. Ist dies der Fall wird kontrolliert, wie viele Tankstellen bis zum Ziel angefahren wurden auf diesem Weg. Ist diese Zahl kleiner als die in „$shortest“ gespeicherte, so wird diese Zahl in „$shortest“ gespeichert und alle vorherigen Wege aus „$tankstellen“ gelöscht. Zuletzt wird dann der aktuelle Weg in „$tankstellen“ gespeichert. Sollte hingegen die Anzahl der auf dem aktuellen Weg benötigten Tankstopps dem Wert in „$shortest“ entsprechen, so wird der aktuelle Weg als ein potentiell optimaler Weg zu „$tankstellen“ hinzugefügt.

```
  else
    if(not vorherige_tankstellen.length >= $shortest)
      erreichbar = erreichbare_tankstellen(ind)
      if erreichbar.length > $grenze
        erreichbar = erreichbar.drop(erreichbar.length-$grenze)   #Zum Einhalten der Obergrenze
      end
```

Sollte hingegen das Ziel noch nicht erreicht sein, so wird zunächst kontrolliert, ob noch nicht so viele Tankstellen angefahren wurden, wie im kürzesten bisher gefundenen Fall. Anschließend werden in diesem Fall mit der zuvor beschriebenen Methode „erreichbare_tankstellen“ alle Tankstellen bestimmt, welche sich von der derzeitigen Position aus erreichen lassen. Werden dabei mehr Tankstellen zurückgegeben, als die Obergrenze zulässt, so werden vom Anfang des Arrays aus so viele Tankstellen entfernt, dass die Grenze eingehalten wird.

```
      erreichbar.reverse_each do |i|
        test(i, vorherige_tankstellen + [ind])
      end
    end
  end
end
```

Zuletzt folgt dann ein rekursiver Aufruf für alle erreichbaren Tankstellen innerhalb der Obergrenze von der aktuellen Position aus. Ist die Obergrenze groß genug, sodass nie Wege die möglich wären nicht ausprobiert werden können, so ist die berechnete Lösung optimal, da für sie dann ausnahmslos alle Wege ausprobiert wurden. Ansonsten steigt die Genauigkeit der Lösung, je höher die Obergrenze ist.
Am Ende befinden sich jedenfalls alle möglichen Wege mit der Minimalanzahl an Tankstellen, die erreicht werden konnte, in dem Array „$tankstellen“.

Die Methode „tanken“ ist dann abschließend dazu da, um für die berechneten Wege zu bestimmen, wie viel jeweils wo genau getankt werden muss, um möglichst niedrige Kosten zu haben. Dabei wird je eine Folge an Tankstellen die zum Ziel führt übergeben und am Ende werden in zwei Arrays für alle Tankstellen die zu tankenden Liter („liter“) und die dazugehörigen Kosten („zu_bezahlen“) zurückgegeben.

```
def tanken(tankstellen)
    i = 0
    fuellung = $f
    liter = Array.new
    zu_bezahlen = Array.new
```

Zu Beginn werden mehrer Variablen definiert: „i“ ist ein Zähler, um alle übergeben bekommenen Tankstellen auf dem Weg abzuarbeiten. „fuellung“ wird zunächst als die Startfüllung des Fahrzeugs festgelegt, da dies die Füllung an der Startposition ist. Zudem werden die oben beschriebenen Arrays „liter“ und „zu_bezahlen“ definiert.

```
    while(i < tankstellen.length - 1) do
        distanz = $d[tankstellen[i+1]].to_f - $d[tankstellen[i]].to_f
        fuellung = fuellung.round(6)
```

Anschließend beginnt eine Schleife, in welcher immer die Tankstellen „i“ und „i+1“ untersucht werden. In dieser Schleife wird zunächst die Distanz zwischen den beiden zuvor genannten Tankstellen bestimmt. Zudem wird „fuellung“ auf sechs Nachkommastellen gerundet, da keine der zu berechnenden Zahlen mehr Nachkommastellen haben sollten und sich so der Fehler durch das Rechnen mit Float Werten beseitigen lässt.
Anschließend wird zwischen zwei Fällen unterschieden: ist die „i“-te oder die „i+1“-te Tankstelle die billigere Tankstelle.

```
        if $p[tankstellen[i]] > $p[tankstellen[i+1]]
            zu_tanken = (($v * distanz / 100) - fuellung).round(6)
            if(zu_tanken >= 0)
                liter.push(zu_tanken + 0.0)
                zu_bezahlen.push((zu_tanken*$p[tankstellen[i]]).round)
                fuellung = 0
            else
                liter.push(0.0)
                zu_bezahlen.push(0)
                fuellung = -zu_tanken
            end
```

Kostet ein Liter bei der „i+1“-ten Tankstelle weniger als bei der „i“-ten, so wird nur so viel bei der „i“-ten getankt, dass Punktgenau die „i+1“-te Tankstelle erreicht werden kann. Der letzte Teil dieses Abschnitts stellt dabei eine Abfangbedingung für die Startposition dar, weil hier nicht getankt werden kann und ein negativer Wert an zu tankenden Litern entsteht, aufgrund der Startfüllung.

```
        else
            if($p[tankstellen[i+1]] == Float::INFINITY)
                liter.push((($v * distanz / 100) - fuellung).round(6))
                zu_bezahlen.push((liter[-1]*$p[tankstellen[i]]).round)
                fuellung = 0
            else
                liter.push(($g-fuellung).round(6))
                zu_bezahlen.push((liter[-1]*$p[tankstellen[i]]).round)
                fuellung = ($g - distanz / 100 * $v).round(6)
            end
```

Andernfalls wird bei der „i“-ten Tankstelle soviel getankt, wie es die Tankgröße zulässt. Auch hier gibt einen Sonderfall: dieser trifft ein, wenn die „i+1“-te „Tankstelle“ in Wirklichkeit das Ziel ist. In diesem Fall muss der Tank nicht komplett gefüllt werden, da es nach Aufgabenstellung genügt mit einem leeren Tank am Ziel anzukommen. Folglich wird hier auch nur genau so viel getankt, dass dies möglich wird. An dieser Stelle sei anzumerken, dass die Kosten für die zu tankenden Liter gerundet werden auf glatte Cent Beträge, da sich nicht weniger als ein Cent bezahlen lässt.

```
      end
      i += 1
    end
    liter.push(0.0)
    zu_bezahlen.push(0)
    return liter, zu_bezahlen
end
```

Abschließend folgt dann nur noch das Hochzählen von „i“ für die Schleife, sowie das jeweilige Anhängen einer Null an die Arrays „liter“ und „zu_bezahlen“ für das Ziel (da sonst die Startposition sowie alle Tankstellen einen Wert haben, nicht aber die Zielposition). Dies erleichtert die anschließende Ausgabe (siehe Source Code Zeile 124 bis 126), da so alle Arrays die selbe Länge haben.

Abschließend folgen dann nur noch der Aufruf der zuvor beschriebenen Methoden sowie das Ermitteln des günstigsten Weges in „$tankstellen“. Dabei wird jeder Weg in „$tankstellen“ einmal an die Methode „tanken“ übergeben und alle Werte des zweiten zurückgegebenen Arrays („zu_bezahlen“) addiert. So werden die Gesamtkosten berechnet. Dabei wird dann immer der Weg mit den niedrigsten bisher gefundenen Gesamtkosten zwischen gespeichert, bis alle Wege ausprobiert wurden. Am Ende wird dann nur noch dieser Weg mit allen dazugehörenden Informationen ausgegeben.

Beispiel/Demonstration:

Wie zuvor angesprochen lassen sich die Beispiele 1, 3 und 4 schnell optimal Lösen. Dies wird dadurch erkennbar, dass für eine beliebig hohe Obergrenze das Programm dennoch schnell zu einer Lösung kommt. In anderen Worten es gibt nicht unglaublich viele Tankstellen in diesen Beispielen, von denen aus man unglaublich viele andere Tankstellen erreichen kann. Dementsprechend ist die hier gefundene Lösung jeweils definitiv optimal. Die genauen Ergebnisse zu diesen Beispielen befinden sich in der Datei „Beispiele 1, 3 und 4.txt“ im Ordner des Programms. Die Mindestanzahlen an anzufahrenden Tankstellen sowie die Gesamtkosten auf dem günstigsten Weg mit dieser Anzahl an Tankstellen einmal zusammengefasst:

fahrt1.txt: 2 Tankstellen | 8170 Cent
fahrt3.txt: 1 Tankstelle | 9920 Cent
fahrt4.txt: 2 Tankstellen | 23688 Cent

Für die Beispiele 2 und 5 ist es mir nicht endgültig möglich zu sagen, dass ich die optimale Lösung erreicht habe. Ich gehe jedoch davon aus, dass meine Ergebnisse optimal sein könnten oder sehr nah an die optimalen Lösungen herankommen, da ich das ganze mit immer größer werdender Obergrenze ausprobiert habe, bis bei beiden Beispielen sich das Ergebnis trotz größerer Obergrenze nicht mehr verbesserte. Da die höchste getestete Obergrenze allerdings bereits zu einem minutenlangem Rechnen führte, war es mir nicht möglich das Program mit beliebig hoher Obergrenze zu testen und somit zu bestätigen, dass es sich bei den gefundenen Lösungen um die bestmöglichen handelt. In den Dateien „Beispiel 2.txt“ und „Beispiel 5.txt“ befinden sich sämtliche Ausgaben für eine Obergrenze von 3 bis 8 zu den Beispielen 2 und 5. Die Ergebnisse wie oben kurz zusammengefasst bei einer Obergrenze von 8 für die Beispiele 2 und 5:

fahrt2.txt: 9 Tankstellen | 253033 Cent
fahrt5.txt: 9 Tankstellen | 249432 Cent

Alle weiteren Informationen finden sich in den .txt-Dateien im Ordner des Programms.