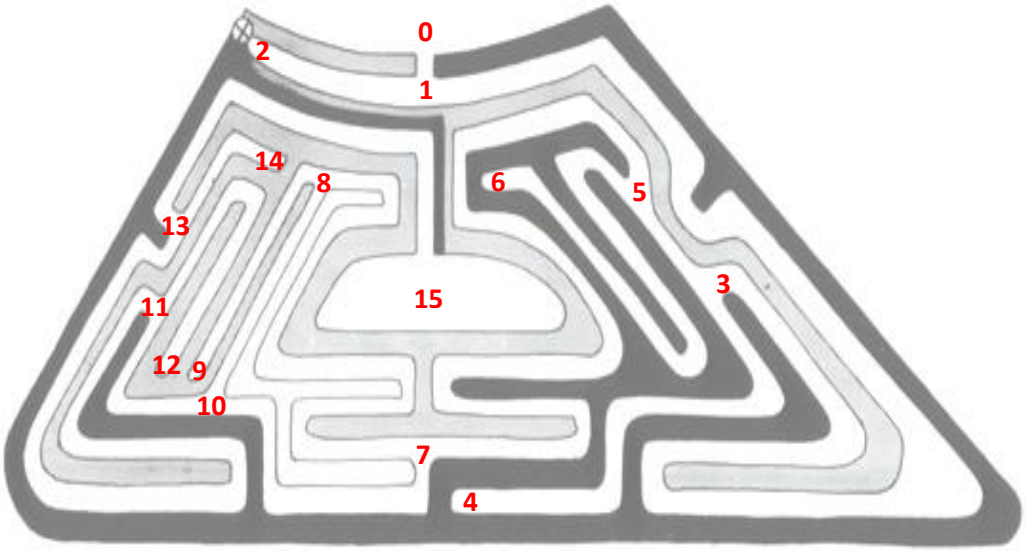
Übung 10 Abgabe 03.07.2019

Aufgabe 1 – Weg aus dem Labyrinth 9 Punkte

Die folgenden Abbildungen zeigen das berühmte Labyrinth von Schloss Hampton Court bei London:



a) Erstellen Sie mit Hilfe der Knotennummern aus der schematischen Darstellung die Graphenrepräsentation durch Adjazenzlisten. Sortieren Sie die einzelnen Adjazenzlisten dabei *absteigend*. Geben Sie den Graphen und die weiteren Teilaufgaben im File `maze.py` ab. *2 Punkte*

b) Implementieren Sie eine Funktion `way_out(graph,startnode,targetnode)`, die den Tiefensuchalgorithmus aus der Vorlesung so modifiziert, dass er für gegebenen Start- und Zielknoten den Weg aus dem Labyrinth bestimmt. Bei jedem Schritt soll der Algorithmus die Nummer des erreichten Knotens ausgeben. Eine Sackgasse soll mit der Ausgabe "dead end" markiert werden. Die auf dem Rückweg aus der Sackgasse erreichten Knoten werden mit dem Zusatz "(backtrack)" versehen, bis es wieder vorwärts geht. Am Ende soll der String "target reached" und die Gesamtzahl der Sackgassen, in die der Algorithmus gelaufen ist, ausgegeben werden. Modifizieren Sie die Funktion `way_out` so, dass sie die besuchten Knoten in Preorder zurückgibt. *3 Punkte*

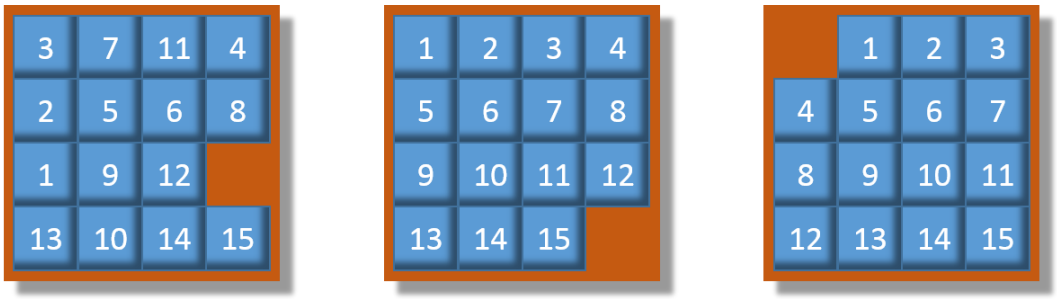
c) Benutzen Sie einen Stack, um den rekursiven Algorithmus aus b) in einen iterativen Algorithmus (also mit einer Schleife statt Rekursion) zu transformieren. Implementieren Sie *4 Punkte*

die Funktion `way_out_stack(graph, startnode, endnode)` mit dem gleichen Verhalten wie `way_out()` bis auf die "backtrack" Meldungen. Tipp: Damit die Knoten in derselben Reihenfolge besucht werden wir bei b), müssen sie in geschickter Reihenfolge in den Stack eingefügt werden.

Modifizieren Sie die Funktion `way_out_stack` so, dass sie die besuchten Knoten in Preorder zurückgibt. Schreiben Sie UnitTests die die Preorder von `way_out` und `way_out_stack` vergleichen.

Aufgabe 2 – Schiebe-Puzzle 13 Punkte

Sie kennen sicherlich das Schiebepuzzle, bei dem man eine zufällige Ausgangsstellung sortieren muss, indem man in jedem Zug ein Zahlenfeld auf die Lücke verschiebt:



Ausgangsstellung Zielstellung A Zielstellung B

Es gibt 16! ≈ $2.1*10^{13}$ Ausgangsstellungen. Genau die Hälfte davon können in die Zielstellung A, die andere Hälfte in die Zielstellung B überführt werden, aber keine in beide. Die Stellung links lässt sich in 19 Zügen nach A transformieren. Für die schwierigsten Stellungen benötigt man 80 Züge bis zum Ziel. Mit einem einfachen Python-Programm wie in dieser Aufgabe kann man in akzeptabler Zeit Lösungswege mit bis zu 20 Zügen finden. Geben Sie Ihren Code im File `schiebepuzzle.py` ab.

a) Die aktuelle Stellung wird in einem Array der Länge 16 repräsentiert, das die Zahlen und das Leerzeichen enthält. Schreiben Sie eine Funktion `print_pos(p)`, die die Stellung p formatiert auf die Konsole ausgibt: *2 Punkte*

```
>>> p = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ' ']
>>> print_pos(p)   # p enthält hier die Zielstellung A
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15
```

b) Erzeugen Sie lösbare Ausgangsstellungen, indem Sie die Zielstellung A mittels Zufallszahlen verwürfeln. Schreiben Sie eine Funktion `shuffle_pos(A,N)`, die von Zielstellung A beginnend *N* zufällige Züge ausführt und die resultierende Stellung zurückgibt. Die möglichen Züge "up" (verschiebe das Feld unterhalb der Lücke nach oben), "left" (verschiebe das Feld rechts der Lücke nach links), "down" und "right" werden dabei durch Zufallszahlen 0…3 ausgewählt. Wenn die Lücke sich am Rand des Spielfelds befindet, sind allerdings nicht alle Züge erlaubt (in Stellung A sind z.B. nur "right" und "down" möglich). Lösen Sie dieses Problem durch *rejection sampling*: Wenn die gewählte Zufallszahl einen unerlaubten Zug bezeichnet, wird sie ignoriert und eine neue gezogen. Ignorieren Sie gleichfalls Zufallszahlen, die den vorhergehenden Zug wieder rückgängig machen (es ist also z.B. verboten, nach "up" sofort "down" auszuführen). *4 Punkte*

c) Man kann die kürzeste Zugfolge zum Ziel mittels Breitensuche finden. Dabei bezeichnet jeder Knoten des Graphen eine Stellung. Stellungen, die durch einen einzigen Zug ineinander übergehen, werden durch Kanten verbunden. Allerdings kann man diesen Graphen nicht *7 Punkte*

vollständig erzeugen, weil er mit 16! Knoten viel zu groß wäre. Deshalb muss der Standardalgorithmus aus der Vorlesung modifiziert werden:

- Man erzeugt die Knoten des Graphen *on the fly*, also erst dann, wenn sie in der Schleife über die Nachbarn tatsächlich benötigt werden. Die Suche wird bei einer gegebenen Suchtiefe abgebrochen, falls bis dahin keine Lösung gefunden wurde.
- Das Array `parents` muss jetzt vom Typ `dict` sein und enthält anfangs nur die Ausgangsstellung. Wird eine Stellung p erstmals erreicht, legt man für sie einen Eintrag in `parents` an. Als Schlüssel wird dabei die Stringdarstellung `str(p)` verwendet (`p` kann nicht direkt als Schlüssel dienen, weil Python für Arrays keine Hashfunktion definiert). Der zugehörige Wert ist der Vaterknoten, also die Stellung, von wo aus der aktuelle Knoten erreicht wurde (dies ist nützlich, um am Ende den Lösungsweg zurückverfolgen und ausdrucken zu können).

Implementieren Sie die Funktion `solve_bfs(p, maxlevel)`, die nach Lösungen mit maximal `maxlevel` Zügen sucht. Wird eine Lösung gefunden, sollen die Anzahl der Züge und der Lösungsweg ausgedruckt werden, also alle Stellungen von der Ausgangsstellung bis zum Ziel. Andernfalls wird nur die Ausgangsstellung mit der Bemerkung `"unsolved"` ausgedruckt. Testen Sie, dass für die Ausgangsstellung im obigen Bild tatsächlich 19 Züge benötigt werden. Erzeugen Sie mit der Funktion `shuffle_pos()` aus b) weitere Ausgangsstellungen und geben Sie die Lösungen dafür aus.

Bitte laden Sie Ihre Lösung bis zum 3.07.2019 um 12:00 Uhr auf Moodle hoch.