Pri LP relaksaciji lahko gledamo dve stvari:

vrednost ciljne funkcije (ki je navzdol omejena z optimalno vrednostjo) in pa množico vozlišč, ki imajo vrednost strogo večjo od 0.5 (tako bo v tej množici nastopalo kvečjemu eno vozlišče izmed krajišč vsake povezave - gre torej za neodvisno množico in kot tako z dopustno rešitev CLP). Bolj elegantno lahko to množico dobiš tako:

sez = [k for k, v in vozlisca.items() if v > 0.5]

(podobno lahko seveda narediš tudi pri CLP).

Kar imenuješ lokalno iskanje je v bistvu le randomiziran požrešni algoritem. Ideja lokalnega iskanja je ta, da obstoječo rešitev poskusiš izboljševati - začneš z neko rešitvijo (lahko iz požrešnega algoritma), nato pa poskusiš to rešitev izboljševati (npr. odstraniš nekaj vozlišč in poskusiš spet dodajati, v upanju, da bo morda mogoče dodati večje število vozlišč). Pri osnovnem lokalnem iskanju (hill climbing) novo rešitev vzameš, če je boljša od prejšnje; obstajajo pa tudi druge metahevristike, npr. simulirano ohlajanje (simulated annealing), kjer z neko verjetnostjo (v odvisnosti od števila koraka) neko rešitev vzameš tudi, če je slabša od prejšnje (seveda je ob tem potrebno hraniti tudi dotedanjo najboljšo rešitev).

~~Priporočam sicer, da namesto s seznamom delaš z množico - inicializiraš jo torej kot~~

~~seznam\_neod = set()~~

~~dodajaš pa z add namesto append. Tako bo preverjanje vsebovanosti hitrejše (pričakovano konstantno namesto linearnega). Potem bo hitrejše tudi preverjanje, ali ima trenutno vozlišče že soseda v množici:~~

~~if not any(u in seznam\_neod for u in graf[v]): # ali je kateri od sosedov vozlišča v v množici?~~

~~seznam\_neod.add(v)~~

Svetujem še, naj gre glavna zanka skozi premešan seznam vozlišč (dobiš ga z graf.vertices(), nato pa na njem uporabiš random.shuffle()) - tako bo možnih več vrstnih redov kot le določitev prvega vozlišča (tega tudi ne rabiš posebej dodajati - pa tudi preverjanje, ali vozlišče že je v množici postane nepotrebno).