Solutions for Assignment 2.

1.

a) Initial state: two arbitrary 8-puzzle states.
Successor function: one move on an unsolved puzzle.
Goal test: both puzzles in goal state.
Path cost: 1 per move.

b) Each puzzle has $9!/2$ reachable states (remember that half the states are unreachable). The joint state space has $(9!*9!/4)$ states.

c) This is like backgammon; expectiminimax works.

d) Actually the statement in the question is not true (it applies to a previous version of part (c) in which the opponent is just trying to prevent you from winning in that case, the coin tosses will eventually allow you to solve one puzzle without interruptions). For the game described in (c), consider a state in which the coin has come up heads, say, and you get to work on a puzzle that is 2 steps from the goal. Should you move one step closer? If you do, your opponent wins if he tosses heads; or if he tosses tails, you toss tails, and he tosses heads; or any sequence where both toss tails n times and then he tosses heads. So his probability of winning is at least $1/2 + 1/8 + 1/32 + ... = 2/3$. So it seems you're better off moving away from the goal. (There's no way to stay the same distance from the goal.) This problem unintentionally seems to have the same kind of solution as suicide tic-tactoe with passing.

2.
If every position n levels 0,1,2…,m – 1 of a game tree satisfying the conditions of having exactly b successors (branch), for some fixed constant b, then the alpha-beta procedure examines exactly $b^{m/2} + b^{m/2} - 1$. (this is also the low bound of alpha-beta procedure, so moves have to be optimal in order to reach the low bound).

Proof:

There are $b^{m/2}$ sequences $a_1, a_2...a_m$, with $1 \leq a_i \leq b$, for all i, such that $a_i = 1$ for all odd values of i; there are $b^{m/2}$ such sequences with $a_i = b$ for all even values of i; and we subtract 1 for sequence 1...1 which was counted twice since m can be even or odd.

Note that sequence $a_1, a_2...a_m$ is assigned to every position on level m. Root node corresponds to the empty sequence.

Thus, the time complexity is $O(2b^{m/2})$ when all moves are optimal.

3.
Description and Pseudocode:

Define “P” as particular instance of the problem to be solved.
Define c = candidate solution at root c (means from the root).
Define update(c,P) as accepting P into candidate c. return a new c’.
Domain = \{0,1,2,...,8,9\}
Define CSP(P,c) returns a boolean value to check one assignment

1. Backtracking (different order from the book, but same concept)
   backtrack(c)
   In order choose P from Domain and loop through
   If CSP(P,c) is false then return; - Partial candidate c is not worth completing
   else c’ = update (c,P) (for first extension)
   While c’ is not complete do
     backtrack(c’)
     c’ = update(c’,P)

Note that this format of Pseudocode differs from the one in the book (want to show a different view), but the concept is same as in this problem:

Input a candidate c
Choose a value P from the domain (assign one letter to a digit value from the domain, in order, no repeat)
Validate the new candidate c after assign P, if the constraint satisfied then accept new c as c’, otherwise cut.
Once the candidate c is complete (all letters have been assigned with numbers from the domain) and it’s a valid solution, return
If candidate c is not complete, call backtrack(c) to keep going.

2. MRV + Backtracking
   You can add one simple additional rule when assigning value from the domain, which is excluding the number from the domain has been assigned to previous value (minimum remaining value). Modification to the Pseudocode is small:

   Additional Definition
   Define unassignedDomain as the original domain exclude those numbers have been assigned.

   unassignedDomain = Domain;
   backtrack (c)
   In order choose P from unassignedDomain and loop through
If CSP(P,c) is false then return
else c' = update(c,P) (for first extension)

```python
unassignedDomain = unassignedDomain - P
```

While c' is not complete do
backtrack(c')
c' = update(c',P)

3. MRV + Forward checking + Backtracking:
Not all valid forward checking functions are required; one example forward-checking function can be described as below:
In the same digits number: A,B,C. For example in tens digit:
XXAX – XBX = XCX
If A and B have valid assigned values from the domain, C can be predicted according your defined function. For example, A-B = C or A-B = C+1 (carry 1). Therefore, let's say we have A = 5, B = 3, C can be only chosen from 1 or 2. If A = 4, B = 2, C can be 2 or 1 according to the defined forward-checking function, but obviously 2 will be eliminated according to the CSP. So, if we combine MRV with Forward-Checking, 2 will never be considered since 2 is excluded from the Domain after it’s assigned. Therefore, Backtracking + MRV + Forward-checking will be faster than either Backtracking+ MRV or Backtracking+ forward checking.

Define function updateUnassignedDomain(X, unassignedDomain), returns a new unassignedDomainDict;

```python
unassignedDomainDict = Dictionary[X,Domain]; (fully mapped initially)
backtrack (c)
    In order choose P from unassignedDomainDict[p] and loop through
    If CSP(P,c) is false then return
    else c' = update (c,P) (for first extension)
        unassignedDomainDict = updateUnassignedDomain
        (X,unassignedDomainDict, P)
    While c’ is not complete do
        backtrack(c’)
        c’ = update(c’,P)
```

UpdateUnassignedDomain (X, unassignedDomain, P)
Find same digits combinations from X and loop through.
Forward-Checking(combination) – return key-value pairs which are valid
Delete Key-Value pair from the unassignedDomainDict
Return updated unassignedDomainDict.